O'REILLY®

TURING 图灵程序设计丛书



C#并发编程经典实例

Concurrency in C# Cookbook

[美] Stephen Cleary 著

相银初 译

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者介绍

相银初

1996年毕业于复旦大学，长期从事软件开发和项目管理工作，涉及C++、C#、Oracle、Linux等技术，也从事软件类图书的翻译工作。


图灵程序设计丛书

C#并发编程经典实例

Concurrency in C# Cookbook

[美] Stephen Cleary 著
相银初 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

C#并发编程经典实例 / (美) 克利里 (Cleary,S.)
著 ; 相银初译. -- 北京 : 人民邮电出版社, 2015.1
(图灵程序设计丛书)
ISBN 978-7-115-37427-1

Ⅰ. ①C… Ⅱ. ①克… ②相… Ⅲ. ①C语言－程序设
计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第260737号

内 容 提 要

本书全面讲解 C# 并发编程技术，侧重于 .NET 平台上较新、较实用的方法。全书分为几大部分：首先介绍几种并发编程技术，包括异步编程、并行编程、TPL 数据流、响应式编程；然后阐述一些重要的知识点，包括测试技巧、互操作、取消并发、函数式编程与 OOP、同步、调度；最后介绍了几个实用技巧。全书共包含 70 多个有配套源码的实用方法，可用于服务器程序、桌面程序和移动应用的开发。

本书适合具有 .NET 基础，希望学习最新并发编程技术的开发人员阅读。

◆ 著　　　　[美] Stephen Cleary
译　　　　相银初
责任编辑　李松峰
执行编辑　李　静　曹静雯
责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：11.75
字数：237千字　　　　2015年 1 月第 1 版
印数：1－3 000册　　　2015年 1 月北京第 1次印刷

著作权合同登记号　图字：01-2014-6523号

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

译者序

关于并发编程的几个误解

关于并发编程，很多人都有一些误解。

误解一：并发就是多线程

实际上多线程只是并发编程的一种形式，在 C# 中还有很多更实用、更方便的并发编程技术，包括异步编程、并行编程、TPL 数据流、响应式编程等。

误解二：只有大型服务器程序才需要考虑并发

服务器端的大型程序要响应大量客户端的数据请求，当然要充分考虑并发。但是桌面程序和手机、平板等移动端应用同样需要考虑并发编程，因为它们是直接面向最终用户的，而现在用户对使用体验的要求越来越高。程序必须能随时响应用户的操作，尤其是在后台处理时（读写数据、与服务器通信等），这正是并发编程的目的之一。

误解三：并发编程很复杂，必须掌握很多底层技术

C# 和 .NET 提供了很多程序库，并发编程已经变得简单多了。尤其是 .NET 4.5 推出了全新的 async 和 await 关键字，使并发编程的代码减少到了最低限度。并行处理和异步开发已经不再是高手们的专利，只要使用本书中的方法，每个开发人员都能写出交互性良好、高效、可靠的并发程序。

本书的特色

本书全面讲解 C# 并发编程技术，侧重于 .NET 平台上较新、较实用的方法。全书分为几大

部分：首先介绍几种并发编程技术，包括异步编程、并行编程、TPL 数据流、响应式编程等；然后是一些重要的知识点，包括测试技巧、互操作、取消并发、函数式编程与 OOP、同步、调度等；最后介绍了几个实用技巧。书中包含 70 多个配有源码的实用方法，可用于服务器程序、桌面程序和移动端应用的开发。

本书填补了一个市场空白：它是一本用最新方法进行并发编程的入门指引和参考书。

本书作者 Stephen Cleary 是美国著名的软件开发者和技术书作家、C# MVP，在 C#/C++/JavaScript 等方面均有丰富的经验。我非常有幸能翻译他的著作。

翻译中的一点感受

过去的十多年我一直在从事软件开发和设计工作。相信国内很多开发人员都和我一样，心中存在着一个疑惑：我国的软件人员很多（绝对数量不会比美国少），但为什么软件技术总体上落后欧美国家那么多？确定翻译《C# 并发编程经典实例》这本书后，我一边仔细阅读原书，一边遵循作者的思路，逐渐发现作者思考问题的一个理念。这就是按软件的不同层次进行明确分工，我只负责我所实现的这个层次，底层技术是为上层服务的，我只负责选择和调用，不管内部的实现过程；同样，我负责的层次为更高一层的软件提供服务，供上层调用，也不需要上层关心我的内部实现。

由此想到，这正好反映出国内开发人员中的一个通病，即分工不够细、技术关注不够精。很多公司和团队在开发时都喜欢大包大揽，从底层到应用层全部自己实现；很多开发人员也热衷于“大而全”地学习技术，试图掌握软件开发中的各种技术，而不是精通某一方面。甚至流行这样一种观点，实现底层软件、写驱动的才是高级开发人员，做上层应用的人仅仅是“码农”。本书作者明确地反对了这种看法，书中强调如何利用好现成的库，而不是全部采用底层技术自己实现。利用现成的库开发出高质量的软件，对技术能力的考验并不低于开发底层库。

感谢

在本书的翻译过程中，得到了图灵公司李松峰老师的支持和帮助，在此表示感谢。由于本人水平有限，书中难免有疏忽和错误，恳请读者朋友们批评指正。

2014 年 10 月于深圳

前言

我觉得封面上的动物（麝香猫）能体现出本书的主题。在看到这个封面之前，我对这种动物一无所知，因此特意查了一下。麝香猫会在天花板和阁楼上随处便溺，并且在最不合时宜的情况下互相打斗发出很大的噪音，因此被认为是一种害兽。它们肛门处的气味腺会分泌一种令人作呕的分泌物。在动物保护分类中，麝香猫属于“无危物种”，这相当于说“人们可以随意捕杀，没人会在乎”。麝香猫喜欢吃咖啡果，并且吃完咖啡豆之后不消化，又排泄出来。世界上最贵的咖啡之一——猫屎咖啡，就是用麝香猫排泄出的咖啡豆制造的。美国特种咖啡协会称“这种咖啡味道好极了”。

这些特征使麝香猫成为代表并发和多线程开发的完美吉祥物。软件开发新手会非常讨厌并发和多线程，它们会让原本整洁的代码变得乱七八糟。竞态条件（race condition）和其他莫名其妙的原因会导致程序严重崩溃（经常在实际产品或演示程序中出现）。有些人甚至声称“多线程是魔鬼”，并且完全不使用并发编程。有少数开发人员已经对并发编程产生兴趣，并毫不畏惧地使用它。但大多数开发人员曾被并发编程搞晕，并且留下了不好的印象。

然而，并发性正在成为现代程序的一个必备特性。今天的软件用户要求程序界面在任何时候都不能停止响应；另外，服务器应用的规模变得越来越大。并发编程顺应了这两种变化趋势。

幸好，已经有很多现代的程序库，使并发编程变得比以前简单多了！并行处理和异步开发，不再是高手们的专利。这些程序库使用更高层次的抽象化，让每一个开发人员都能开发出具有很好的响应性和可扩展性的程序。如果在并发编程还非常困难的时候你曾经感到困惑，我建议你用现代工具重新试一下。我们不能说并发编程很容易，但确实不像以前那么难了。

本书读者对象

本书面向希望学习最新并发编程方法的开发人员。你需要熟练掌握 .NET 开发，包括泛型集合（generic collection）、枚举（enumerable）和 LINQ。你不需要具备任何多线程或异步

开发的知识。本书介绍新的、更安全、更易使用的程序库，因此如果你已有这方面的经验，读这本书也会有所帮助。

并发编程适用于所有程序。不管是桌面程序、移动应用还是服务器应用，现在并发性几乎是所有程序的必备特性。利用本书提供的方法，可以提高用户界面的响应速度和服务器应用的可扩展性。现在，并发编程已经非常普遍，对一个专业开发人员来说，掌握并使用有关技术非常必要。

本书写作初衷

在我职业生涯的早期，我费了很大力气学习多线程开发。几年后，我又费了很大力气学习异步开发。尽管那些经验很有价值，但我仍然很希望当时就能有今天的工具和资源。尤其是现在的 .NET 语言对 async 和 await 的支持，实在太棒了。

然而，现在大多数介绍并发编程的图书和资料都是从最底层概念开始讲起。那些书用大量篇幅讲解有关多线程和序列化的基本概念，并且把较高级的技术内容放到最后。我觉得这么做的原因有两个。首先，很多像我这样的并发编程开发人员确实是从底层技术学起，费劲地学习这些老技术。其次，很多书是多年前出版的，现在出现了新技术，改版时就把新技术的内容放到书的末尾。

我觉得那种做法有些落伍。本书只介绍进行并发编程的最新方法。这并不是说，理解全部底层概念没用。我进入大学学习编程时，有一门课程需要利用少量的门电路来组建一个虚拟的 CPU，另一门课程则需要用汇编语言进行开发。在我的职业生涯里，从来没有设计过 CPU，也很少写汇编程序，但是理解那些基础知识对我的日常工作仍然很有帮助。但最好是从更高级的抽象概念开始学习，我学的第一种编程语言也不是汇编语言。

本书填补了一项市场空白：它是一本用最新方法进行并发编程的入门指引和参考书。本书包含了几种类型的并发编程，包括并行、异步和响应式编程（reactive programming）。至于并发编程的老技术，有关图书和网上资料有很多，本书不再介绍。

内容速览

本书既是一本入门指引，也是一本快捷参考书。全书分为几个部分。

- 第 1 章，简要介绍本书涉及的几种并发编程类型：并行、异步、响应式编程以及数据流。
- 第 2 章至第 5 章，更详细地介绍这几种并发编程类型。
- 其余章节，分别讲解并发编程的各个方面，也可作为解决常见问题时的参考书。

即使你已经熟悉某些类型的并发编程，建议你还是要读第 1 章，至少略读一下。

网上资料

本书较全面地介绍了几种并发编程类型，尽可能包含所有相关知识点，但不管怎样，一本书无法包罗万象。要更全面地了解并发编程相关技术，推荐学习下面的资料。

并行编程方面，推荐阅读 *Parallel Programming with Microsoft .NET*（Microsoft Press），英文原书电子版可以从网上下载。可惜这本书的内容有点过时了。例如，“future 模式”部分应该改用异步编程，“流水线”（pipeline）部分应该改用任务 TPL 数据流。

异步编程方面，推荐阅读 MSDN，特别是“Task-based Asynchronous Pattern”这篇文档。

TPL 数据流方面，推荐阅读微软发布的“Introduction to TPL Dataflow”文档。

网络上，响应式扩展（Rx）程序库越来越流行了，并且它本身还在继续发展。在我看来，学习 Rx 最好的资料是 Lee Campbell 写的 *Introduction to Rx*。

排版规范

本书使用了以下排版规范。

- 楷体
  用于表示新术语。

- 等宽字体
  用于表示程序代码，或者段落中提及的代码元素（变量名、函数名、数据库、数据类型、环境变量、程序语句、关键字）。

- 等宽粗体
  表示需要用户逐字输入的命令或者其他文本。

- 等宽斜体
  表示需要根据用户提供的内容，或者根据上下文替换掉的文字。



这个图标表示提示、建议或注解。



这个图标表示警告或提醒。

Safari® Books Online



Safari Books Online（http://www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920030171.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

本书的出版离不开很多人的帮助。

首先，我要感谢上帝和救世主耶稣基督。成为基督徒是我此生最重要的决定！如果你想了解这方面的更多信息，欢迎通过我的个人网站（http://stephencleary.com）联系我。

其次，我要感谢我的家人，感谢他们容许我拿出那么多本该陪伴他们的时间写书。开始动笔时，有从事写作的朋友告诉我："你将有一年的时间无法陪伴家人！"当时我还以为他们是在开玩笑。我白天工作，晚上和周末用来写作，对此我的妻子 Mandy、孩子 SD 和 Emma 都非常理解。太感谢你们了，我爱你们！

当然，下面这些人极大地提高了本书质量：编辑 Brian MacDonald、技术评审 Stephen Toub、Petr Onderka（"svick"）和 Nick Paldino（"casperOne"）。如果书中有错误，那全是他们的责任。开个玩笑！他们对内容的调整和修改非常有价值，如果书中还有错误，当然是我自己的责任。

最后，我要感谢 Stephen Toub、Lucian Wischik、Thomas Levesque 和 Lee Campbell，我是从他们那里学到的有关技术。他们是 Stack Overflow 和 MSDN 论坛的成员，也是我的家乡密歇根州及周边地区软件研讨会的参与者。我有幸成为软件开发社区的一员，如果这本书具有一些价值，那只是因为那么多人给我指明方向。感谢大家！

第 1 章

并发编程概述

优秀软件的一个关键特征就是具有并发性。过去的几十年，我们可以进行并发编程，但是难度很大。以前，并发性软件的编写、调试和维护都很难，这导致很多开发人员为图省事放弃了并发编程。新版 .NET 中的程序库和语言特征，已经让并发编程变得简单多了。随着 Visual Studio 2012 的发布，微软明显降低了并发编程的门槛。以前只有专家才能做并发编程，而今天，每一个开发人员都能够（而且应该）接受并发编程。

1.1　并发编程简介

首先，我来解释几个贯穿本书始终的术语。先来介绍并发。

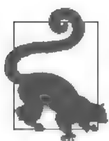
- *并发*
  同时做多件事情。

这个解释直接表明了并发的作用。终端用户程序利用并发功能，在输入数据库的同时响应用户输入。服务器应用利用并发，在处理第一个请求的同时响应第二个请求。只要你希望程序同时做多件事情，你就需要并发。几乎每个软件程序都会受益于并发。

在编写本书时（2014 年），大多数开发人员一看到“并发”就会想到“多线程”。对这两个概念，需要做一下区分。

- *多线程*
  并发的一种形式，它采用多个线程来执行程序。

从字面上看，多线程就是使用多个线程。本书后续章节将介绍，多线程是并发的一种形式，但不是唯一的形式。实际上，直接使用底层线程类型在现代程序中基本不起作用。比起老式的多线程机制，采用高级的抽象机制会让程序功能更加强大、效率更高。因此，本书将尽量不涉及一些过时的技术。书中所有多线程的方法都采用高级类型，而不是 `Thread` 或 `BackgroundWorker`。



一旦你输入 `new Thread()`，那就糟糕了，说明项目中的代码太过时了。

但是，不要认为多线程已经彻底被淘汰了！因为线程池要求多线程继续存在。线程池存放任务的队列，这个队列能够根据需要自行调整。相应地，线程池产生了另一个重要的并发形式：*并行处理*。

- *并行处理*
  把正在执行的大量的任务分割成小块，分配给多个同时运行的线程。

为了让处理器的利用效率最大化，并行处理（或并行编程）采用多线程。当现代多核 CPU 执行大量任务时，若只用一个核执行所有任务，而其他核保持空闲，这显然是不合理的。并行处理把任务分割成小块并分配给多个线程，让它们在不同的核上独立运行。

并行处理是多线程的一种，而多线程是并发的一种。在现代程序中，还有一种非常重要但很多人还不熟悉的并发类型：*异步编程*。

- *异步编程*
  并发的一种形式，它采用 future 模式或回调（callback）机制，以避免产生不必要的线程。

一个 future（或 promise）类型代表一些即将完成的操作。在 .NET 中，新版 future 类型有 `Task` 和 `Task<TResult>`。在老式异步编程 API 中，采用回调或事件（event），而不是 future。异步编程的核心理念是异步操作：启动了的操作将会在一段时间后完成。这个操作正在执行时，不会阻塞原来的线程。启动了这个操作的线程，可以继续执行其他任务。当操作完成时，会通知它的 future，或者调用回调函数，以便让程序知道操作已经结束。

异步编程是一种功能强大的并发形式，但直至不久前，实现异步编程仍需要特别复杂的代码。VS2012 支持 `async` 和 `await`，这让异步编程变得几乎和同步（非并发）编程一样容易。

并发编程的另一种形式是*响应式编程*（reactive programming）。异步编程意味着程序启动一个操作，而该操作将会在一段时间后完成。响应式编程与异步编程非常类似，不过它是基

于异步事件（asynchronous event）的，而不是异步操作（asynchronous operation）。异步事件可以没有一个实际的“开始”，可以在任何时间发生，并且可以发生多次，例如用户输入。

- 响应式编程
  一种声明式的编程模式，程序在该模式中对事件做出响应。

如果把一个程序看作一个大型的状态机，则该程序的行为便可视为它对一系列事件做出响应，即每换一个事件，它就更新一次自己的状态。这听起来很抽象和空洞，但实际上并非如此。利用现代的程序框架，响应式编程已经在实际开发中广泛使用。响应式编程不一定是并发的，但它与并发编程联系紧密，因此本书介绍了响应式编程的基础知识。

通常情况下，一个并发程序要使用多种技术。大多数程序至少使用了多线程（通过线程池）和异步编程。要大胆地把各种并发编程形式进行混合和匹配，在程序的各个部分使用合适的工具。

1.2 异步编程简介

异步编程有两大好处。第一个好处是对于面向终端用户的 GUI 程序：异步编程提高了响应能力。我们都遇到过在运行时会临时锁定界面的程序，异步编程可以使程序在执行任务时仍能响应用户的输入。第二个好处是对于服务器端应用：异步编程实现了可扩展性。服务器应用可以利用线程池满足其可扩展性，使用异步编程后，可扩展性通常可以提高一个数量级。

现代的异步 .NET 程序使用两个关键字：`async` 和 `await`。`async` 关键字加在方法声明上，它的主要目的是使方法内的 `await` 关键字生效（为了保持向后兼容，同时引入了这两个关键字）。如果 `async` 方法有返回值，应返回 `Task<T>`；如果没有返回值，应返回 `Task`。这些 task 类型相当于 future，用来在异步方法结束时通知主程序。



不要用 `void` 作为 `async` 方法的返回类型！ `async` 方法可以返回 `void`，但是这仅限于编写事件处理程序。一个普通的 `async` 方法如果没有返回值，要返回 `Task`，而不是 `void`。

有了上述背景知识，我们来快速看一个例子：

```
async Task DoSomethingAsync()
{
    int val = 13;

    // 异步方式等待 1 秒
    await Task.Delay(TimeSpan.FromSeconds(1));

    val *= 2;
```

```
    // 异步方式等待 1 秒
    await Task.Delay(TimeSpan.FromSeconds(1));

    Trace.WriteLine(val);
}
```

和其他方法一样，async 方法在开始时以同步方式执行。在 async 方法内部，await 关键字对它的参数执行一个异步等待。它首先检查操作是否已经完成，如果完成了，就继续运行（同步方式）。否则，它会暂停 async 方法，并返回，留下一个未完成的 task。一段时间后，操作完成，async 方法就恢复运行。

一个 async 方法是由多个同步执行的程序块组成的，每个同步程序块之间由 await 语句分隔。第一个同步程序块在调用这个方法的线程中运行，但其他同步程序块在哪里运行呢？情况比较复杂。

最常见的情况是，用 await 语句等待一个任务完成，当该方法在 await 处暂停时，就可以捕捉上下文（context）。如果当前 SynchronizationContext 不为空，这个上下文就是当前 SynchronizationContext。如果当前 SynchronizationContext 为空，则这个上下文为当前 TaskScheduler。该方法会在这个上下文中继续运行。一般来说，运行 UI 线程时采用 UI 上下文，处理 ASP.NET 请求时采用 ASP.NET 请求上下文，其他很多情况下则采用线程池上下文。

因此，在上面的代码中，每个同步程序块会试图在原始的上下文中恢复运行。如果在 UI 线程中调用 DoSomethingAsync，这个方法的每个同步程序块都将在此 UI 线程上运行。但是，如果在线程池线程中调用，每个同步程序块将在线程池线程上运行。

要避免这种错误行为，可以在 await 中使用 ConfigureAwait 方法，将参数 continueOnCapturedContext 设为 false。接下来的代码刚开始会在调用的线程里运行，在被 await 暂停后，则会在线程池线程里继续运行：

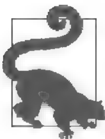
```
async Task DoSomethingAsync()
{
    int val = 13;

    // 异步方式等待 1 秒
    await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);

    val *= 2;

    // 异步方式等待 1 秒
    await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);

    Trace.WriteLine(val.ToString());
}
```



最好的做法是，在核心库代码中一直使用 ConfigureAwait。在外围的用户界面代码中，只在需要时才恢复上下文。

关键字 await 不仅能用于任务，还能用于所有遵循特定模式的 awaitable 类型。例如，Windows Runtime API 定义了自己专用的异步操作接口。这些接口不能转化为 Task 类型，但确实遵循了可等待的（awaitable）模式，因此可以直接使用 await。这种 awaitable 类型在 Windows 应用商店程序中更加常见，但是在大多数情况下，await 使用 Task 或 Task<T>。

有两种基本的方法可以创建 Task 实例。有些任务表示 CPU 需要实际执行的指令，创建这种计算类的任务时，使用 Task.Run（如需要按照特定的计划运行，则用 TaskFactory.StartNew）。其他的任务表示一个通知（notification），创建这种基于事件的任务时，使用 TaskCompletionSource<T>。大部分 I/O 型任务采用 TaskCompletionSource<T>。

使用 async 和 await 时，自然要处理错误。在下面的代码中，PossibleExceptionAsync 会抛出一个 NotSupportedException 异常，而 TrySomethingAsync 方法可很顺利地捕捉到这个异常。这个捕捉到的异常完整地保留了栈轨迹，没有人为地将它封装进 TargetInvocationException 或 AggregateException 类：

```
async Task TrySomethingAsync()
{
    try
    {
        await PossibleExceptionAsync();
    }
    catch(NotSupportedException ex)
    {
        LogException(ex);
        throw;
    }
}
```

一旦异步方法抛出（或传递出）异常，该异常会放在返回的 Task 对象中，并且这个 Task 对象的状态变为“已完成”。当 await 调用该 Task 对象时，await 会获得并（重新）抛出该异常，并且保留着原始的栈轨迹。因此，如果 PossibleExceptionAsync 是异步方法，以下代码就能正常运行：

```
async Task TrySomethingAsync()
{
    // 发生异常时，任务结束。不会直接抛出异常。
    Task task = PossibleExceptionAsync();

    try
    {
        //Task 对象中的异常，会在这条 await 语句中引发
```

```
            await task;
        }
        catch(NotSupportedException ex)
        {
            LogException(ex);
            throw;
        }
    }
```

关于异步方法，还有一条重要的准则：你一旦在代码中使用了异步，最好一直使用。调用异步方法时，应该（在调用结束时）用 await 等待它返回的 task 对象。一定要避免使用 Task.Wait 或 Task<T>.Result 方法，因为它们会导致死锁。参考一下下面这个方法：

```
async Task WaitAsync()
{
    // 这里 awati 会捕获当前上下文……
    await Task.Delay(TimeSpan.FromSeconds(1));
    // ……这里会试图用上面捕获的上下文继续执行
}

void Deadlock()
{
    // 开始延迟
    Task task = WaitAsync();

    // 同步程序块，正在等待异步方法完成
    task.Wait();
}
```

如果从 UI 或 ASP.NET 的上下文调用这段代码，就会发生死锁。这是因为，这两种上下文每次只能运行一个线程。Deadlock 方法调用 WaitAsync 方法，WaitAsync 方法开始调用 delay 语句。然后，Deadlock 方法（同步）等待 WaitAsync 方法完成，同时阻塞了上下文线程。当 delay 语句结束时，await 试图在已捕获的上下文中继续运行 WaitAsync 方法，但这个步骤无法成功，因为上下文中已经有了一个阻塞的线程，并且这种上下文只允许同时运行一个线程。这里有两个方法可以避免死锁：在 WaitAsync 中使用 ConfigureAwait(false)（导致 await 忽略该方法的上下文），或者用 await 语句调用 WaitAsync 方法（让 Deadlock 变成一个异步方法）。



如果使用了 async，最好就一直使用它。

若想更全面地了解关于异步编程的知识，可参阅 Alex Davies（O'Reilly）编写的 *Async in C# 5.0*，这本书非常不错。另外，微软公司有关异步编程的在线文档也很不错，建议你至少读一读"async overview"和"Task-based Asynchronous Pattern(TAP) overview"这两篇。如果要深入了解，官方 FAQ 和博客上也有大量的信息。

1.3 并行编程简介

如果程序中有大量的计算任务，并且这些任务能分割成几个互相独立的任务块，那就应该使用并行编程。并行编程可临时提高 CPU 利用率，以提高吞吐量，若客户端系统中的 CPU 经常处于空闲状态，这个方法就非常有用，但通常并不适合服务器系统。大多数服务器本身具有并行处理能力，例如 ASP.NET 可并行地处理多个请求。某些情况下，在服务器系统中编写并行代码仍然有用（如果你知道并发用户数量会一直是少数）。但通常情况下，在服务器系统上进行并行编程，将降低本身的并行处理能力，并且不会有实际的好处。

并行的形式有两种：*数据并行*（data parallelism）和*任务并行*（task parallelim）。数据并行是指有大量的数据需要处理，并且每一块数据的处理过程基本上是彼此独立的。任务并行是指需要执行大量任务，并且每个任务的执行过程基本上是彼此独立的。任务并行可以是动态的，如果一个任务的执行结果会产生额外的任务，这些新增的任务也可以加入任务池。

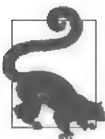
实现数据并行有几种不同的做法。一种做法是使用 `Parallel.ForEach` 方法，它类似于 `foreach` 循环，应尽可能使用这种做法。在 3.1 节将会详细介绍 `Parallel.ForEach` 方法。`Parallel` 类也提供 `Parallel.For` 方法，这类似于 `for` 循环，当数据处理过程基于一个索引时，可使用这个方法。下面是使用 `Parallel.ForEach` 的代码例子：

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees)
{
    Parallel.ForEach(matrices, matrix => matrix.Rotate(degrees));
}
```

另一种做法是使用 PLINQ（Parallel LINQ），它为 LINQ 查询提供了 AsParallel 扩展。跟 PLINQ 相比，Parallel 对资源更加友好，Parallel 与系统中的其他进程配合得比较好，而 PLINQ 会试图让所有的 CPU 来执行本进程。Parallel 的缺点是它太明显。很多情况下，PLINQ 的代码更加优美。PLINQ 在 3.5 节有详细介绍：

```
IEnumerable<bool> PrimalityTest(IEnumerable<int> values)
{
    return values.AsParallel().Select(val => IsPrime(val));
}
```

不管选用哪种方法，在并行处理时有一个非常重要的准则。



每个任务块要尽可能的互相独立。

只要任务块是互相独立的，并行性就能做到最大化。一旦你在多个线程中共享状态，就必须以同步方式访问这些状态，那样程序的并行性就变差了。第 11 章将详细讲述同步。

有多种方式可以控制并行处理的输出。可以把结果存在某些并发集合，或者对结果进行聚合。聚合在并行处理中很常见，Parallel 类的重载方法，也支持这种 map/reduce 函数。关于聚合的详细内容在 3.2 节。

下面讲任务并行。数据并行重点在处理数据，任务并行则关注执行任务。

Parallel 类的 Parallel.Invoke 方法可以执行“分叉 / 联合”（fork/join）方式的任务并行。3.3 节将详细介绍这个方法。调用该方法时，把要并行执行的委托（delegate）作为传入参数：

```
void ProcessArray(double[] array)
{
    Parallel.Invoke(
        () => ProcessPartialArray(array, 0, array.Length / 2),
        () => ProcessPartialArray(array, array.Length / 2, array.Length)

    );
}

void ProcessPartialArray(double[] array, int begin, int end)
{
    // CPU 密集型的操作……
}
```

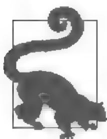
现在 Task 这个类也被用于异步编程，但当初它是为了任务并行而引入的。任务并行中使用的一个 Task 实例表示一些任务。可以使用 Wait 方法等待任务完成，还可以使用 Result 和 Exception 属性来检查任务执行的结果。直接使用 Task 类型的代码比使用 Parallel 类要复杂，但是，如果在运行前不知道并行任务的结构，就需要使用 Task 类型。如果使用动态并行机制，在开始处理时，任务块的个数是不确定的，只有继续执行后才能确定。通常情况下，一个动态任务块要启动它所需的所有子任务，然后等待这些子任务执行完毕。为实现这个功能，可以使用 Task 类型中的一个特殊标志：TaskCreationOptions.AttachedToParent。动态并行机制在 3.4 节中详述。

跟数据并行一样，任务并行也强调任务块的独立性。委托（delegate）的独立性越强，程序的执行效率就越高。在编写任务并行程序时，要格外留意下闭包（closure）捕获的变量。记住闭包捕获的是引用（不是值），因此可以在结束时以不明显地方式地分享这些变量。

对所有并行处理类型来讲，错误处理的方法都差不多。由于操作是并行执行的，多个异常就会同时发生，系统会把这些异常封装在 AggregateException 类里，在程序中抛给代码。这一特点对所有方法都是一样的，包括 Parallel.ForEach、Paralle.lInvoke、Task.Wait 等。AggregateException 类型有几个实用的 Flatten 和 Handle 方法，用来简化错误处理的代码：

```
try
{
    Parallel.Invoke(() => { throw new Exception(); },
        () => { throw new Exception(); });
}
catch (AggregateException ex)
{
    ex.Handle(exception =>
    {
        Trace.WriteLine(exception);
        return true; // “已经处理”
    });
}
```

通常情况下，没必要关心线程池处理任务的具体做法。数据并行和任务并行都使用动态调整的分割器，把任务分割后分配给工作线程。线程池在需要的时候会增加线程数量。线程池线程使用工作窃取队列（work-stealing queue）。微软公司为了让每个部分尽可能高效，做了很多优化。要让程序得到最佳的性能，有很多参数可以调节。只要任务时长不是特别短，采用默认设置就会运行得很好。



任务不要特别短，也不要特别长。

如果任务太短，把数据分割进任务和在线程池中调度任务的开销会很大。如果任务太长，线程池就不能进行有效的动态调整以达到工作量的平衡。很难确定“太短”和“太长”的判断标准，这取决于程序所解决问题的类型以及硬件的性能。根据一个通用的准则，只要没有导致性能问题，我会让任务尽可能短（如果任务太短，程序性能会突然降低）。更好的做法是使用 `Parallel` 类型或者 PLINQ，而不是直接使用任务。这些并行处理的高级形式，自带有自动分配任务的算法（并且会在运行时自动调整）。

要更深入的了解并行编程，这方面最好的书是 Colin Campbell 等人编写的 *Parallel Programming with Microsoft.NET*（微软出版社）。

1.4　响应式编程简介

跟并发编程的其他形式相比，响应式编程的学习难度较大。如果对响应式编程不是非常熟悉，代码维护相对会更难一点。一旦你学会了，就会发现响应式编程的功能特别强大。响应式编程可以像处理数据流一样处理事件流。根据经验，如果事件中带有参数，那么最好采用响应式编程，而不是常规的事件处理程序。

响应式编程基于“可观察的流”（observable stream）这一概念。你一旦申请了可观察流，就

可以收到任意数量的数据项（OnNext），并且流在结束时会发出一个错误（OnError）或一个“流结束”的通知（OnCompleted）。有些可观察流是不会结束的。实际的接口就像这样：

```
interface IObserver<in T>
{
    void OnNext(T item);
    void OnCompleted();
    void OnError(Exception error);
}

interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}
```

不过，开发人员不需要实现这些接口。微软的 Reactive Extensions（Rx）库已经实现了所有接口。响应式编程的最终代码非常像 LINQ，可以认为它就是“LINQ to events”。下面的代码中，前面是我们不熟悉的操作符（Interval 和 Timestamp），最后是一个 Subscribe，但是中间部分是我们在 LINQ 中熟悉的操作符：Where 和 Select。LINQ 具有的特性，Rx 也都有。Rx 在此基础上增加了很多它自己的操作符，特别是与时间有关的操作符：

```
Observable.Interval(TimeSpan.FromSeconds(1))
    .Timestamp()
    .Where(x => x.Value % 2 == 0)
    .Select(x => x.Timestamp)
    .Subscribe(x => Trace.WriteLine(x));
```

上面的代码中，首先是一个延时一段时间的计数器（Interval），随后、后为每个事件加了一个时间戳（Timestamp）。接着对事件进行过滤，只包含偶数值（Where），选择了时间戳的值（Timestamp），然后当每个时间戳值到达时，把它输入调试器（Subscribe）。如果没有理解上述新的操作符（例如 Interval），不要紧，我们会在后面讲述。现在只要记住这是一个 LINQ 查询，与你以前见过的 LINQ 查询很类似。主要区别在于：LINQ to Object 和 LINQ to Entity 使用“拉取”模式，LINQ 的枚举通过查询拉出数据。而 LINQ to event（Rx）使用“推送”模式，事件到达后就自行穿过查询。

可观察流的定义和其订阅是互相独立的。上面最后一个例子与下面的代码等效：

```
IObservable<DateTimeOffset> timestamps =
    Observable.Interval(TimeSpan.FromSeconds(1))
    .Timestamp()
    .Where(x => x.Value % 2 == 0)
    .Select(x => x.Timestamp);
timestamps.Subscribe(x => Trace.WriteLine(x));
```

一种常规的做法是把可观察流定义为一种类型，然后将其作为 IObservable<T> 资源使用。其他类型可以订阅这些流，或者把这些流与其他操作符组合，创建另一个可观察流。

Rx 的订阅也是一个资源。Subscribe 操作符返回一个 IDisposable，即表示订阅完成。当你响应了那个可观察流，就得处理这个订阅。

对于 hot observable（热可观察流）和 cold observable（冷可观察流）这两种对象，订阅的做法各有不同。一个 hot observable 对象是指一直在发生的事件流，如果在事件到达时没有订阅者，事件就丢失了。例如，鼠标的移动就是一个 hot observable 对象。old observable 对象是始终没有输入事件（不会主动产生事件）的观察流，它只会通过启动一个事件队列来响应订阅。例如，HTTP 下载是一个 cold observable 对象，只有在订阅后才会发出 HTTP 请求。

同样，所有 Subscribe 操作符都需要有处理错误的参数。前面的例子没有错误处理参数。下面则是一个更好的例子，在可观察流发生错误时，它能正确处理：

```
Observable.Interval(TimeSpan.FromSeconds(1))
    .Timestamp()
    .Where(x => x.Value % 2 == 0)
    .Select(x => x.Timestamp)
    .Subscribe(x => Trace.WriteLine(x),
        ex => Trace.WriteLine(ex));
```

在进行 Rx 实验性编程时，Subject<T> 这个类型很有用。这个“subject”就像手动实现一个可观察流。可以在代码中调用 OnNext、OnError 和 OnCompleted，这个 subject 会把这些调用传递给订阅者。Subject<T> 用于实验时效果非常不错，但在实际产品开发时，应该使用第 5 章介绍的操作符。

Rx 的操作符非常多，本书只介绍了一部分。想了解关于 Rx 的更多信息，建议阅读优秀的在线图书 *Introduction to Rx*。

1.5　数据流简介

TPL 数据流很有意思，它把异步编程和并行编程这两种技术结合起来。如果需要对数据进行一连串的处理，TPL 数据流就很有用。例如，需要从一个 URL 上下载数据，接着解析数据，然后把它与其他数据一起做并行处理。TPL 数据流通常作为一个简易的管道，数据从管道的一端进入，在管道中穿行，最后从另一端出来。不过，TPL 数据流的功能比普通管道要强大多了。对于处理各种类型的网格（mesh），在网格中定义分叉（fork）、连接（join）、循环（loop）的工作，TPL 数据流都能正确地处理。当然了，大多数时候 TPL 数据流网格还是被用作管道。

数据流网格的基本组成单元是数据流块（dataflow block）。数据流块可以是目标块（接收数据）或源块（生成数据），或两者皆可。源块可以连接到目标块，创建网格。连接的具体内容在 4.1 节介绍。数据流块是半独立的，当数据到达时，数据流块会试图对数据进行处理，并且把处理结果推送给下一个流程。使用 TPL 数据流的常规方法是创建所有的块，

再把它们链接起来，然后开始在一端填入数据。然后，数据会自行从另一端出来。再强调一次，数据流的功能比这要强大得多，数据穿过的同时，可能会断开连接、创建新的块并加入到网格，不过这是非常高级的使用场景。

目标块带有缓冲区，用来存放收到的数据。因此，在还来不及处理数据的时候，它仍能接收新的数据项，这就让数据可以持续地在网格上流动。在有分叉的情况下，一个源块链接了两个目标块，这种缓冲机制就会产生问题。当源块有数据需要传递下去时，它会把数据传给与它链接的块，并且一次只传一个数据。默认情况下，第一个目标块会接收数据并缓存起来，而第二个目标块就收不到任何数据。解决这个问题的方法是把目标块设置为“非贪婪”模式，以限制缓冲区的数量，这部分将在 4.4 节介绍。

如果某些步骤出错，例如委托在处理数据项时抛出异常，数据流块就会出错。数据流块出错后就会停止接收数据。默认情况下，一个块出错不会摧毁整个网格。这让程序有能力重建部分网格，或者对数据重新定向。然而这是一个高级用法。通常来讲，你是希望这些错误通过链接传递给目标块。数据流也提供这个选择，唯一比较难办的地方是当异常通过链接传递时，它就会被封装在 AggregateException 类中。因此，如果管道很长，最后异常的嵌套层次会非常多，这时就可以使用 AggregateException.Flatten 方法：

```
try
{
    var multiplyBlock = new TransformBlock<int, int>(item =>
    {
        if (item == 1)
            throw new InvalidOperationException("Blech.");
        return item * 2;
    });
    var subtractBlock = new TransformBlock<int, int>(item => item - 2);
    multiplyBlock.LinkTo(subtractBlock,
        new DataflowLinkOptions { PropagateCompletion = true });

    multiplyBlock.Post(1);
    subtractBlock.Completion.Wait();
}
catch (AggregateException exception)
{
    AggregateException ex = exception.Flatten();
    Trace.WriteLine(ex.InnerException);
}
```

数据流错误的处理方法将在 4.2 节详细介绍。

数据流网格给人的第一印象是与可观察流非常类似，实际上它们确实有很多共同点。网格和流都有“数据项”这一概念，数据项从网格或流的中间穿过。还有，网格和流都有“正常完成”（表示没有更多数据需要接收时发出的通知）和“不正常完成”（在处理数据中发生错误时发出的通知）这两个概念。但是，Rx 和 TPL 数据流的性能并不相同。如果执行

需要计时的任务，最好使用 Rx 的 observable 对象，而不是数据流块。如果进行并行处理，最好使用数据流块，而不是 Rx 的 observable 对象。从概念上说，Rx 更像是建立回调函数：observable 对象中的每个步骤都会直接调用下一步。相反，数据流网格中的每一块都是互相独立的。Rx 和 TPL 数据流有各自的应用领域，也有一些交叉的领域。另一方面，Rx 和 TPL 数据流也非常适合同时使用。Rx 和 TPL 数据流的互操作性将在 7.7 节详细介绍。

最常用的块类型有 `TransformBlock<TInput, TOutput>`（与 LINQ 的 `Select` 类似）、`TransformManyBlock<TInput, Toutput>`（与 LINQ 的 `SelectMany` 类似）和 `ActionBlock<T>`（为每个数据项运行一个委托）。要了解 TPL 数据流的更多知识，建议阅读 MSDN 的文档和 *Guide to Implementing Custom TPL Dataflow Blocks*。

1.6 多线程编程简介

线程是一个独立的运行单元，每个进程内部有多个线程，每个线程可以各自同时执行指令。每个线程有自己独立的栈，但是与进程内的其他线程共享内存。对某些程序来说，其中有一个线程是特殊的，例如用户界面程序有一个 UI 线程，控制台程序有一个 `main` 线程。

每个 .NET 程序都有一个线程池，线程池维护着一定数量的工作线程，这些线程等待着执行分配下来的任务。线程池可以随时监测线程的数量。配置线程池的参数多达几十个，但是建议采用默认设置，线程池的默认设置是经过仔细调整的，适用于绝大多数现实中的应用场景。

应用程序几乎不需要自行创建新的线程。你若要为 COM interop 程序创建 SAT 线程，就得创建线程，这是唯一需要线程的情况。

线程是低级别的抽象，线程池是稍微高级一点的抽象，当代码段遵循线程池的规则运行时，线程池就会在需要时创建线程。本书介绍的技术抽象级别更高：并行和数据流的处理队列会根据情况遵循线程池运行。抽象级别更高，正确代码的编写就更容易。

基于这个原因，本书根本不介绍 `Thread` 和 `BackgroundWorker` 这两种类型。它们曾经非常流行，但那个时代已经过去了。

1.7 并发编程的集合

并发编程所用到的集合有两类：并发集合和不可变集合。这两种类别的集合将在第 8 章详细介绍。多个线程可以用安全的方式同时更新并发集合。大多数并发集合使用快照（snapshot），当一个线程在增加或删除数据时，另一个线程也能枚举数据。比起给常规集合加锁以保护数据的方式，采用并发集合的方式要高效得多。

不可变集合则有些不同。不可变集合实际上是无法修改的。要修改一个不可变集合，需要

建立一个新的集合来代表这个被修改了的集合。这看起来效率非常低，但是不可变集合的各个实例之间尽可能多地共享存储区，因此实际上效率没想象得那么差。不可变集合的优点之一，就是所有的操作都是简洁的，因此特别适合在函数式代码中使用。

1.8 现代设计

大多数并发编程技术有一个类似点：它们本质上都是函数式（functional）的。这里“functional”的意思不是“实用，能完成任务”[1]，而是把它作为一种基于函数组合的编程模式。如果你接受函数式的编程理念，并发编程的设计就会简单得多。

函数式编程的一个原则就是简洁（换言之，就是避免副作用）。解决方案中的每一个片段都用一些值作为输入，生成一些值作为输出。应该尽可能避免让这些段落依赖于全局（或共享）变量，或者修改全局（或共享）数据结构。不论这个片段是异步方法、并行任务、Rx 操作还是数据流块，都应该这么做。当然了，具体做法迟早会受到计算内容的影响，但如果能用简洁的段落来处理，然后用结果来执行更新，代码就会更加清晰。

函数式编程的另一个原则是不变性。不变性是指一段数据是不能被修改的。在并发编程中使用不可变数据的原因之一，是程序永远不需要对不可变数据进行同步。数据不能修改，这一事实让同步变得没有必要。不可变数据也能避免副作用。在编写本书时（2014 年），虽然不可变数据还没有被广泛接受，但本书中有几节会介绍不可变数据结构。

1.9 技术要点总结

在 .NET 刚推出时，就对异步编程提供了一定的支持。但是异步编程一直是很难的，直到 2012 年 .NET 4.5（同时发布 C# 5.0 和 VB 2012）引入 `async` 和 `await` 这两个关键字。本书中的异步编程方法，将全部采用现代的 `async/await`。同时介绍一些方法，来实现 `async` 和老式异步编程模式的交互。要支持老式平台的话，需要下载 NuGet 包 `Microsoft.Bcl.Async`。



不要在基于 .NET 4.0 的 ASP.NET 代码中使用 `Microsoft.Bcl.Async` 进行异步编程！在 .NET 中，ASP.NET 管道已经进行修改以支持 `async`。对于异步 ASP.NET 项目，必须使用 .NET 4.5 或更高版本。

.NET 4.0 引入了并行任务库（TPL），完全支持数据并行和任务并行。但是一些资源较少的平台（例如手机），通常不支持 TPL。TPL 是 .NET 框架自带的。

Reactive Extensions 团队已经让它尽可能多地支持多种平台。Reactive Extensions 和 `async`、`await` 一样，对所有类型的应用都有好处，包括客户端和服务器端应用。Rx 在 NuGet 包

注 1　英文中“函数式”和“实用”是同一个单词 functional。——译者注

Rx-Main 中。

TPL 数据流库只支持较新的平台，它的官方版本在 NuGet 包 Microsoft.Tpl.Dataflow 中。

并发编程的集合是 .NET 框架的一部分，但是不可变集合在 NuGet 包 Microsoft.Bcl.Immutable 中。表 1-1 列出了各主流平台对各种技术的支持情况。

表1-1：各平台对并发编程的支持

平　　台	async	并行编程	Rx	数据流	并发集合	不可变集合
.NET 4.5	√	√	√	√	√	√
.NET 4.0	√	√	√	×	√	×
Mono iOS/Droid	√	√	√	√	√	√
Windows Store	√	√	√	√	√	√
Windows Phone Apps 8.1	√	√	√	√	√	√
Windows Phone SL 8.0	√	×	√	√	×	√
Windows Phone SL 7.1	√	×	√	×	×	×
Silverlight 5	√	×	√	×	×	×

第 2 章

异步编程基础

本章介绍在异步操作中使用 async 和 await 的基础知识。本章只涉及本质上适合异步的操作，例如 HTTP 请求、数据库指令、Web 服务调用等。

如果要把 CPU 密集型的操作当作异步操作来处理（让它不阻塞 UI 线程），请阅读第 3 章和 7.4 节。另外，本章只涉及只启动一次、结束一次的操作。如果要处理事件流，请阅读第 5 章。

要在老版本的平台上使用 async，需要安装 NuGet 包 Microsoft.Bcl.Async。有些平台本身就支持 async，而有些平台需要安装这个 NuGet 包（见表 2-1）。

表2-1：各平台对async的支持情况

平　台	async
.NET 4.5	√
.NET 4.0	NuGet
Mono iOS/Droid	√
Windows Store	√
Windows Phone Apps 8.1	√
Windows Phone SL 8.0	√
Windows Phone SL 7.1	NuGet
Silverlight 5	NuGet

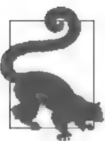
2.1 暂停一段时间

问题

需要让程序（以异步方式）等待一段时间。这在进行单元测试或者实现重试延迟时非常有用。本解决方案也能用于实现简单的超时。

解决方案

Task 类有一个返回 Task 对象的静态函数 Delay，这个 Task 对象会在指定的时间后完成。

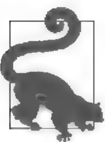


如果程序使用 Microsoft.Bcl.Async 这个 NuGet 库，则 Delay 是 TaskEx 类的成员，而不是 Task 类的成员。

下面的例子用于单元测试，定义了一个异步完成的任务。在模拟一个异步操作时，至少要测试“同步成功”“异步成功”和“异步失败”这三种情况，这一点很重要。下面的例子返回一个 Task 对象，用于“异步成功”测试。

```
static async Task<T> DelayResult<T>(T result, TimeSpan delay)
{
    await Task.Delay(delay);
    return result;
}
```

下一个例子实现了一个简单的指数退避。指数退避是一种重试策略，重试的延迟时间会逐次增加。在访问 Web 服务时，最好的方式就是采用指数退避，它可以防止服务器被太多的重试阻塞。



在实际产品的开发中，建议你采用更周密的方案，例如微软企业库中的瞬间错误处理模块（Transient Error Handling Block）。下面的代码只是一个使用 Task.Delay 的简单例子。

```
static async Task<string> DownloadStringWithRetries(string uri)
{
    using (var client = new HttpClient())
    {
        // 第 1 次重试前等 1 秒，第 2 次等 2 秒，第 3 次等 4 秒。
        var nextDelay = TimeSpan.FromSeconds(1);
        for (int i = 0; i != 3; ++i)
        {
            try
            {
```

```
                return await client.GetStringAsync(uri);
            }
            catch
            {
            }

            await Task.Delay(nextDelay);
            nextDelay = nextDelay + nextDelay;
        }

        // 最后重试一次，以便让调用者知道出错信息。
        return await client.GetStringAsync(uri);
    }
}
```

最后的例子用 Task.Delay 实现一个简单的超时功能。本例中代码的目的是：如果服务在 3 秒内没有响应，就返回 null。

```
static async Task<string> DownloadStringWithTimeout(string uri)
{
    using (var client = new HttpClient())
    {
        var downloadTask = client.GetStringAsync(uri);
        var timeoutTask = Task.Delay(3000);

        var completedTask = await Task.WhenAny(downloadTask, timeoutTask);
        if (completedTask == timeoutTask)
            return null;
        return await downloadTask;
    }
}
```

讨论

Task.Delay 适合用于对异步代码进行单元测试或者实现重试逻辑。要实现超时功能的话，最好使用 CancellationToken。

参阅

2.5 节介绍如何用 Task.WhenAny 来判断哪个任务是首先完成的。

9.3 节介绍用 CancellationToken 实现超时功能的方法。

2.2 返回完成的任务

问题

如何实现一个具有异步签名的同步方法。如果从异步接口或基类继承代码，但希望用同步

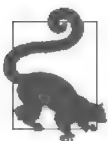
的方法来实现它，就会出现这种情况。对异步代码做单元测试，以及用简单的生成方法存根（stub）或者模拟对象（mock）来产生异步接口，这两种情况下都可使用这种技术。

解决方案

可以使用 Task.FromResult 方法创建并返回一个新的 Task<T> 对象，这个 Task 对象是已经完成的，并有指定的值。

```
interface IMyAsyncInterface
{
    Task<int> GetValueAsync();
}

class MySynchronousImplementation : IMyAsyncInterface
{
    public Task<int> GetValueAsync()
    {
        return Task.FromResult(13);
    }
}
```



如果使用了 Microsoft.Bcl.Async，FromResult 方法就在 TaskEx 类中。

讨论

在用同步代码实现异步接口时，要避免使用任何形式的阻塞操作。在异步方法中进行阻塞操作，然后返回一个完成的 Task 对象，这种做法并不可取。作为一个反例，我们来看一下 .NET 4.5 中 Console 类的文本读取器。Console.In.ReadLineAsync 一定会阻塞调用它的线程，直到它读取完一行文字，然后会返回一个已完成的 Task 对象。这种实现方式并不直观，很多开发人员也觉得很奇怪。一旦异步方法阻塞，调用它的线程就无法启动其他任务，这会干扰程序的并发性，甚至可能产生死锁。

Task.FromResult 只能提供结果正确的同步 Task 对象。如果要让返回的 Task 对象有一个其他类型的结果（例如以 NotImplementedException 结束的 Task 对象），就得自行创建使用 TaskCompletionSource 的辅助方法：

```
static Task<T> NotImplementedAsync<T>()
{
    var tcs = new TaskCompletionSource<T>();
    tcs.SetException(new NotImplementedException());
    return tcs.Task;
}
```

从概念上讲，`Task.FromResult` 只不过是 `TaskCompletionSource` 的一个简化版本，它与上面的代码非常类似。

如果用 `Task.FromResult` 反复调用同一参数，则可考虑用一个实际的 task 变量。例如，可以一次性建立一个结果为 0 的 `Task<int>` 对象，在以后的调用中就不需要创建额外的实例了，这样可减少垃圾回收的次数：

```
private static readonly Task<int> zeroTask = Task.FromResult(0);
static Task<int> GetValueAsync()
{
    return zeroTask;
}
```

参阅

6.1 节介绍异步方法的单元测试。

10.1 节介绍 async 方法的继承。

2.3 报告进度

问题

异步操作执行的过程中，需要展示操作的进度。

解决方案

使用 `IProgress<T>` 和 `Progress<T>` 类型。编写的 async 方法需要有 `IProgress<T>` 参数，其中 `T` 是需要报告的进度类型：

```
static async Task MyMethodAsync(IProgress<double> progress = null)
{
    double percentComplete = 0;
    while (!done)
    {
        ...
        if (progress != null)
            progress.Report(percentComplete);
    }
}
```

调用上述方法的代码：

```
static async Task CallMyMethodAsync()
{
    var progress = new Progress<double>();
```

```
        progress.ProgressChanged += (sender, args) =>
        {
            ...
        };
        await MyMethodAsync(progress);
    }
```

讨论

按照惯例，如果不需要报告进度，IProgress<T> 参数可以是 null，因此在 async 方法中一定要对此进行检查。

需要注意的是，IProgress<T>.Report 方法可以是异步的。这意味着真正报告进度之前，MyMethodAsync 方法会继续运行。基于这个原因，最好把 T 定义为一个不可变类型，或者至少是值类型。如果 T 是一个可变的引用类型，就必须在每次调用 IProgress<T>.Report 时，创建一个单独的副本。

Progress<T> 会在创建时捕获当前上下文，并且在这个上下文中调用回调函数。这意味着，如果在 UI 线程中创建了 Progress<T>，就能在 Progress<T> 的回调函数中更新 UI，即使异步方法是在后台线程中调用 Report 的。

如果一个方法可以报告进度，就该尽量做到可以被取消。

参阅

9.4 节介绍如何实现异步方法的取消功能。

2.4 等待一组任务完成

问题

执行几个任务，等待它们全部完成。

解决方案

框架提供的 Task.WhenAll 方法可以实现这个功能。这个方法的输入为若干个任务，当所有任务都完成时，返回一个完成的 Task 对象：

```
Task task1 = Task.Delay(TimeSpan.FromSeconds(1));
Task task2 = Task.Delay(TimeSpan.FromSeconds(2));
Task task3 = Task.Delay(TimeSpan.FromSeconds(1));

await Task.WhenAll(task1, task2, task3);
```

如果所有任务的结果类型相同，并且全部成功地完成，则 Task.WhenAll 返回存有每个任务执行结果的数组：

```
Task task1 = Task.FromResult(3);
Task task2 = Task.FromResult(5);
Task task3 = Task.FromResult(7);

int[] results = await Task.WhenAll(task1, task2, task3);

// "results" 含有 { 3, 5, 7 }
```

Task.WhenAll 方法有以 IEnumerable 类型作为参数的重载，但建议大家不要使用。只要异步代码与 LINQ 结合，显式的“具体化”序列（即对序列求值，创建集合）就会使代码更清晰：

```
static async Task<string> DownloadAllAsync(IEnumerable<string> urls)
{
    var httpClient = new HttpClient();

    // 定义每一个 url 的使用方法。
    var downloads = urls.Select(url => httpClient.GetStringAsync(url));
    // 注意，到这里，序列还没有求值，所以所有任务都还没真正启动。

    // 下面，所有的 URL 下载同步开始。
    Task<string>[] downloadTasks = downloads.ToArray();
    // 到这里，所有的任务已经开始执行了。

    // 用异步方式等待所有下载完成。
    string[] htmlPages = await Task.WhenAll(downloadTasks);

    return string.Concat(htmlPages);
}
```



如果使用 Microsoft.Bcl.Async 这个 NuGet 库，则 WhenAll 是 TaskEx 类的成员，而不是 Task 类的成员。

讨论

如果有一个任务抛出异常，则 Task.WhenAll 会出错，并把这个异常放在返回的 Task 中。如果多个任务抛出异常，则这些异常都会放在返回的 Task 中。但是，如果这个 Task 在被 await 调用，就只会抛出其中的一个异常。如果要得到每个异常，可以检查 Task.WhenALl 返回的 Task 的 Exception 属性：

```
static async Task ThrowNotImplementedExceptionAsync()
{
    throw new NotImplementedException();
}
```

```
static async Task ThrowInvalidOperationExceptionAsync()
{
    throw new InvalidOperationException();
}

static async Task ObserveOneExceptionAsync()
{
    var task1 = ThrowNotImplementedExceptionAsync();
    var task2 = ThrowInvalidOperationExceptionAsync();

    try
    {
        await Task.WhenAll(task1, task2);
    }
    catch (Exception ex)
    {
        // ex 要么是 NotImplementedException，要么是 InvalidOperationException
        ...
    }
}

static async Task ObserveAllExceptionsAsync()
{
    var task1 = ThrowNotImplementedExceptionAsync();
    var task2 = ThrowInvalidOperationExceptionAsync();

    Task allTasks = Task.WhenAll(task1, task2);
    try
    {
        await allTasks;
    }
    catch
    {
        AggregateException allExceptions = allTasks.Exception;
        ...
    }
}
```

使用 `Task.WhenAll` 时，我一般不会检查所有的异常。通常情况下，只处理第一个错误就足够了，没必要处理全部错误。

参阅

2.5 节介绍等待一批任务中的任意一个完成的方法。

2.6 节介绍等待一批任务完成，并逐个处理完成的任务。

2.8 节介绍对 async 方法的异常处理。

2.5 等待任意一个任务完成

问题

执行若干个任务，只需要对其中任意一个的完成进行响应。这主要用于：对一个操作进行多种独立的尝试，只要一个尝试完成，任务就算完成。例如，同时向多个 Web 服务询问股票价格，但是只关心第一个响应的。

解决方案

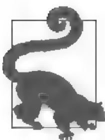
使用 Task.WhenAny 方法。该方法的参数是一批任务，当其中任意一个任务完成时就会返回。作为返回值的 Task 对象，就是那个完成的任务。不要觉得迷惑，这个听起来有点难，但从代码看很容易实现：

```
// 返回第一个响应的 URL 的数据长度。
private static async Task<int> FirstRespondingUrlAsync(string urlA, string urlB)
{
    var httpClient = new HttpClient();

    // 并发地开始两个下载任务。
    Task<byte[]> downloadTaskA = httpClient.GetByteArrayAsync(urlA);
    Task<byte[]> downloadTaskB = httpClient.GetByteArrayAsync(urlB);

    // 等待任意一个任务完成。
    Task<byte[]> completedTask =
        await Task.WhenAny(downloadTaskA, downloadTaskB);

    // 返回从 URL 得到的数据的长度。
    byte[] data = await completedTask;
    return data.Length;
}
```



如果使用 Microsoft.Bcl.Async 这个 NuGet 库，则 WhenAny 是 TaskEx 类的成员，而不是 Task 类的成员。

讨论

Task.WhenAny 返回的 task 对象永远不会以“故障”或“已取消”状态作为结束。该方法的运行结果总是一个 Task 首先完成。如果这个任务完成时有异常，这个异常也不会传递给 Task.WhenAny 返回的 Task 对象。因此，通常需要在 Task 对象完成后继续使用 await。

第一个任务完成后，考虑是否要取消剩下的任务。如果其他任务没有被取消，也没有被继续 await，那它们就处于被遗弃的状态。被遗弃的任务会继续运行直到完成，它们的结果

会被忽略，抛出的任何异常也会被忽略。

使用 Task.WhenAny 可以实现超时功能（例如用 Task.Delay 作为其中的一个任务），但这种做法并不可取。更常见的做法是采用专门有取消功能的超时函数，并且取消功能还有一个好处，就是可以把已经超时的任务彻底取消。

Task.WhenAny 的另一个反模式是处理已完成的任务。一种做法是把所有任务放在一个列表里，在一个任务完成后就把它移除，这种做法看起来好像有道理。问题是这种做法需要执行的时间是 $O(N^2)$，而实际上有时间复杂度为 $O(N)$ 的算法。时间复杂度为 $O(N)$ 的正确算法将在 2.6 节介绍。

参阅

2.4 节介绍异步地等待所有任务完成。

2.6 节介绍等待一批任务完成，并对每个任务进行处理。

9.3 节介绍使用取消标志来实现超时功能。

2.6　任务完成时的处理

问题

正在 await 一批任务，希望在每个任务完成时对它做一些处理。另外，希望在任务一完成就立即进行处理，而不需要等待其他任务。

举个例子，下面的代码启动了 3 个延时任务，然后对每一个进行 await。

```
static async Task<int> DelayAndReturnAsync(int val)
{
    await Task.Delay(TimeSpan.FromSeconds(val));
    return val;
}

// 当前，此方法输出“2”，“3”，“1”。
// 我们希望它输出“1”，“2”，“3”。
static async Task ProcessTasksAsync()
{
    // 创建任务队列。
    Task<int> taskA = DelayAndReturnAsync(2);
    Task<int> taskB = DelayAndReturnAsync(3);
    Task<int> taskC = DelayAndReturnAsync(1);
    var tasks = new[] { taskA, taskB, taskC };

    // 按顺序 await 每个任务。
    foreach (var task in tasks)
```

```
        {
            var result = await task;
            Trace.WriteLine(result);
        }
    }
```

虽然列表中的第二个任务是首先完成的，当前这段代码仍按列表的顺序对任务进行 await。我们希望按任务完成的次序进行处理（例如 Trace.WriteLine），不必等待其他任务。

解决方案

解决这个问题的方法有好几种。本节首先介绍一种推荐大家使用的方法，另一种则将在“讨论部分”给出。

最简单的方案是通过引入更高级的 async 方法来 await 任务，并对结果进行处理，从而重新构建代码。提取出处理过程后，代码就明显简化了。

```
static async Task<int> DelayAndReturnAsync(int val)
{
    await Task.Delay(TimeSpan.FromSeconds(val));
    return val;
}

static async Task AwaitAndProcessAsync(Task<int> task)
{
    var result = await task;
    Trace.WriteLine(result);
}

// 现在，这个方法输出“1”，“2”，“3”。
static async Task ProcessTasksAsync()
{
    // 创建任务队列。
    Task<int> taskA = DelayAndReturnAsync(2);
    Task<int> taskB = DelayAndReturnAsync(3);
    Task<int> taskC = DelayAndReturnAsync(1);
    var tasks = new[] { taskA, taskB, taskC };

    var processingTasks = (from t in tasks
        select AwaitAndProcessAsync(t)).ToArray();

    // 等待全部处理过程的完成。
    await Task.WhenAll(processingTasks);
}
```

上面的代码也可以这么写：

```
static async Task<int> DelayAndReturnAsync(int val)
{
    await Task.Delay(TimeSpan.FromSeconds(val));
    return val;
}
```

```
// 现在，这个方法输出“1”，“2”，“3”。
static async Task ProcessTasksAsync()
{
    // 创建任务队列。
    Task<int> taskA = DelayAndReturnAsync(2);
    Task<int> taskB = DelayAndReturnAsync(3);
    Task<int> taskC = DelayAndReturnAsync(1);
    var tasks = new[] { taskA, taskB, taskC };

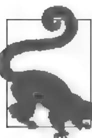
    var processingTasks = tasks.Select(async t =>
    {
        var result = await t;
        Trace.WriteLine(result);
    }).ToArray();

    // 等待全部处理过程的完成。
    await Task.WhenAll(processingTasks);
}
```

重构后的代码是解决本问题最清晰、可移植性最好的方法。不过它与原始代码有一个细微的区别。重构后的代码并发地执行处理过程，而原始代码是一个接着一个地处理。大多数情况下这不会有什么影响，但如果不允许有这种区别，可考虑使用锁（11.2 节介绍）或者后面介绍的可选方案。

讨论

如果上面重构代码的办法不可取，我们还有可选方案。Stephen Toub 和 Jon Skeet 都开发了一个扩展方法，可以让任务按顺序完成，并返回一个任务数组。Setphen Toub 的解决方案见博客文档“Parallel Programming with .NET”（http://t.cn/RhR2V6n），Jon Skeet 的解决方案见他的博客（http://t.cn/RhR2xu9）。



这个扩展方法也可在开源项目 AsyncEx 库（https://nitoasyncex.codeplex.com）找到，它在 NuGet 包 `Nito.AsyncEx`（https://www.nuget.org/packages/Nito.AsyncEx）中。

使用像 `OrderByCompletion` 这样的扩展方法，就能让修改原代码的量降到最低。

```
static async Task<int> DelayAndReturnAsync(int val)
{
    await Task.Delay(TimeSpan.FromSeconds(val));
    return val;
}

// 现在，这个方法输出“1”，“2”，“3”。
static async Task UseOrderByCompletionAsync()
{
```

```
    // 创建任务队列。
    Task<int> taskA = DelayAndReturnAsync(2);
    Task<int> taskB = DelayAndReturnAsync(3);
    Task<int> taskC = DelayAndReturnAsync(1);
    var tasks = new[] { taskA, taskB, taskC };

    // 等待每一个任务完成。
    foreach (var task in tasks.OrderByCompletion())
    {
        var result = await task;
        Trace.WriteLine(result);
    }
}
```

参阅

2.4 节介绍异步地等待一系列任务完成。

2.7 避免上下文延续

问题

在默认情况下，一个 async 方法在被 await 调用后恢复运行时，会在原来的上下文中运行。如果是 UI 上下文，并且有大量的 async 方法在 UI 上下文中恢复，就会引起性能上的问题。

解决方案

为了避免在上下文中恢复运行，可让 await 调用 ConfigureAwait 方法的返回值，参数 continueOnCapturedContext 设为 false：

```
async Task ResumeOnContextAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));

    // 这个方法在同一个上下文中恢复运行。
}

async Task ResumeWithoutContextAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);

    // 这个方法在恢复运行时，会丢弃上下文。
}
```

讨论

如果在 UI 线程上运行的延续任务（continuation）太多，会导致性能上的问题。因为使系统

变慢的方法不止一个，所以这种类型的性能问题是很难诊断的。而且随着程序复杂性的增加，UI 性能会因为“成千上万的剪纸”（在 UI 线程中有太多任务切换，就像剪纸）而变慢。

真正的问题是，在 UI 线程中有多少延续任务，才算是太多？这没有固定的答案，不过微软公司的 Lucian Wischik 公布了一个 WinRT 团队的指导标准（http://t.cn/RhR2KGi）：每秒 100 个左右尚可，但每秒 1000 个左右就太多了。

最好在一开始就避免这个问题。对于每一个 async 方法，如果它没有必要恢复到原来的上下文，就要使用 ConfigureAwait。这么做没有什么坏处。

还有一个好点子，就是在编写 async 代码时特别注意上下文。通常一个 async 方法要么需要上下文（处理 UI 元素或 ASP.NET 请求 / 响应），要么需要摆脱上下文（执行后台指令）。如果一个 asnync 方法的一部分需要上下文、一部分不需要上下文，则可考虑把它拆分为两个（或更多）async 方法。这种做法有利于更好地将代码组织成不同层次。

参阅

第 1 章简要介绍了异步编程。

2.8 处理async Task方法的异常

问题

对任何设计来说，异常处理都是一个关键的部分。只考虑成功情况的设计是很简单的，但是正确的设计必须要能处理异常。还好，处理 async Task 方法的异常是很简单、很直观的。

解决方案

可以用简单的 try/catch 来捕获异常，和同步代码使用的方法一样：

```
static async Task ThrowExceptionAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    throw new InvalidOperationException("Test");
}

static async Task TestAsync()
{
    try
    {
        await ThrowExceptionAsync();
    }
    catch (InvalidOperationException)
    {
```

```
        }
    }
```

在 async Task 方法中引发的异常，存放在返回的 Task 对象中。只有当 Task 对象被 await 调用时，才会引发异常：

```
    static async Task ThrowExceptionAsync()
    {
        await Task.Delay(TimeSpan.FromSeconds(1));
        throw new InvalidOperationException("Test");
    }

    static async Task TestAsync()
    {
        // 抛出异常并将其存储在 Task 中。
        Task task = ThrowExceptionAsync();
        try
        {
            // Task 对象被 await 调用，异常在这里再次被引发。
            await task;
        }
        catch (InvalidOperationException)
        {
            // 这里，异常被正确地捕获。
        }
    }
```

讨论

从 async Task 方法中抛出的异常会被捕获并放在返回的 Task 对象中。因为 async void 方法没有返回 Task 对象，无法存放异常，所以做法就会不同。我们将在另一节介绍这方面的内容。

当 await 调用有异常的 Task 对象时，对象里的第一个异常会重新抛出。若你对重新抛出异常的问题比较熟悉的话，就会担心栈轨迹是否会出错。请放心：异常重新抛出时，原始的栈轨迹会被正确地保存。

这种安排看起来有些复杂，但是当一切复杂性紧密配合后，就只需要很简单的代码。一般情况下，需要把异常从异步方法中传递出来。为此，只需要用 await 调用异步方法返回的 Task 对象，异常就会很顺利地传递出来。

有些情况下（例如 Task.WhenAll），一个 Task 对象包含多个异常，而 await 只会重新抛出第一个。2.4 节有一个处理所有异常的例子。

参阅

2.4 节介绍如何等待多个任务。

2.9 节介绍如何捕获 async void 方法的异常。

6.2 节介绍如何对 async Task 方法抛出的异常做单元测试。

2.9　处理async void方法的异常

问题

需要处理从 async void 方法传递出来的异常。

解决方案

没有什么好的办法。如果可能的话，方法的返回类型不要用 void，把它改为 Task。某些情况下这是不可能的，例如，需要对一个 ICommand 的实现（必须返回 void）做单元测试。这种情况下，可以为 Execute 方法提供一个返回 Task 类型的重载，就像这样：

```
sealed class MyAsyncCommand : ICommand
{
    async void ICommand.Execute(object parameter)
    {
        await Execute(parameter);
    }

    public async Task Execute(object parameter)
    {
        ... // 这里实现异步操作。
    }

    ... // 其他成员（CanExecute 等）。
}
```

最好不要从 async void 方法传递出异常。如果必须使用 async void 方法，可考虑把所有代码放在 try 块中，直接处理异常。

处理 async void 方法的异常还有一个办法。一个异常从 async void 方法中传递出来时，会在 SynchronizationContext 中引发出来。当 async void 方法启动时，SynchronizationContext 就处于激活状态。如果系统运行环境支持 SynchronizationContext，通常就可以在全局范围内处理这些顶层的异常。例如，WPF 有 Application.DispatcherUnhandledException，WinRT 有 Application.UnhandledException，ASP.NET 有 Application_Error。

通过控制 SynchronizationContext，也可以处理从 async void 方法传出的异常。自己编写 SynchronizationContext 的工作量太大，可以使用免费的 NuGet 库 Nito.AsyncEx，里面有 AsyncContext 类。AsyncContext 可以在没有自带 SynchronizationContext 的场合发挥作用，例如控制台程序、Win32 服务程序。下面的例子是在控制台程序中使用 AsyncContext，其

中 async 方法不返回 Task，但 AsyncContext 仍能在 async void 方法中起作用：

```
static class Program
{
    static int Main(string[] args)
    {
        try
        {
            return AsyncContext.Run(() => MainAsync(args));
        }
        catch (Exception ex)
        {
            Console.Error.WriteLine(ex);
            return -1;
        }
    }

    static async Task<int> MainAsync(string[] args)
    {
        ...
    }
}
```

讨论

推荐使用 async Task 而不是 async void，原因之一就是返回 Task 的方法更容易测试。至少要用 Task 方法重载 void 方法，那样可以提供便于测试的 API 外壳。

如果你确实需要自己编写 SynchronizationContext 类（例如 AsyncContext），千万不要把这个 SynchronizationContext 类放到不属于你的线程上。作为通用的准则，不要在已经有 SynchronizationContext 的线程上（比如 UI 或 ASP.NET request 线程）安装这个类，也不要在线程池线程上放 SynchronizationContext。属于你的线程有控制台程序的主线程，还有你自己创建的所有线程。



AsyncContext 类在 NuGet 包 Nito.AsyncEx 中。

参阅

2.8 节介绍 async Task 方法的异常处理。

6.3 节介绍 async void 方法的单元测试。

第3章

并行开发的基础

本章介绍并行编程模式。并行编程用于分解计算密集型的任务片段，并将它们分配给多个线程。这些并行处理方法只适用于计算密集型的任务。如果有需要异步操作的任务（例如I/O 密集型任务），而你希望它能并行运行，请看第 2 章，特别是 2.4 节。

本章介绍的并行处理概念是任务并行库（TPL）的一部分。它只在 .NET 框架中实现，其他平台不一定适用（见表 3-1）。

表3-1：支持TPL的平台

平　台	支持并行编程
.NET 4.5	✓
.NET 4.0	✓
Mono iOS/Droid	✓
Windows Store	✓
Windows Phone Apps 8.1	✓
Windows Phone SL 8.0	×
Windows Phone SL 7.1	×
Silverlight 5	×

3.1　数据的并行处理

问题

有一批数据，需要对每个元素进行相同的操作。该操作是计算密集型的，需要耗费一定的时间。

解决方案

Parallel 类型有专门为此设计的 ForEach 方法。下面的例子使用了一批矩阵，对每一个矩阵都进行旋转：

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees)
{
    Parallel.ForEach(matrices, matrix => matrix.Rotate(degrees));
}
```

在某些情况下需要尽早结束这个循环，例如发现了无效值时。下面的例子反转每一个矩阵，但是如果发现有无效的矩阵，则中断循环：

```
void InvertMatrices(IEnumerable<Matrix> matrices)
{
    Parallel.ForEach(matrices, (matrix, state) =>
    {
        if (!matrix.IsInvertible)
            state.Stop();
        else
            matrix.Invert();
    });
}
```

更常见的情况是可以取消并行循环，这与结束循环不同。结束（stop）循环是在循环内部进行的，而取消（cancel）循环是在循环外部进行的。例如，点击“取消”按钮可以取消一个 CancellationTokenSource，以取消并行循环，方法如下：

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees,
    CancellationToken token)
{
    Parallel.ForEach(matrices,
        new ParallelOptions { CancellationToken = token },
        matrix => matrix.Rotate(degrees));
}
```

需要注意的是，每个并行任务可能都在不同的线程中运行，因此必须保护对共享的状态。下面的例子反转每个矩阵并统计无法反转的矩阵数量：

```
// 注意，这不是最高效的实现方式。
// 只是举个例子，说明用锁来保护共享状态。
int InvertMatrices(IEnumerable<Matrix> matrices)
{
    object mutex = new object();
    int nonInvertibleCount = 0;
    Parallel.ForEach(matrices, matrix =>
    {
        if (matrix.IsInvertible)
        {
            matrix.Invert();
```

```
            }
            else
            {
                lock (mutex)
                {
                    ++nonInvertibleCount;
                }
            }
        });
        return nonInvertibleCount;
    }
```

讨论

Parallel.ForEach 方法可以对一系列值进行并行处理。还有一个类似的解决方案，就是使用 PLINQ（并行 LINQ）。PLINQ 的大部分功能和 Parallel 类一样，并且采用与 LINQ 类似的语法。Parallel 类和 PLINQ 之间有一个区别：PLINQ 假设可以使用计算机内所有的 CPU 核，而 Parallel 类则会根据 CPU 状态的变化动态地调整。

Parallel.ForEach 是并行版本的 foreach 循环。Parallel 类也提供了并行版本的 for 循环，即 Parallel.For 方法。如果有多个数组的数据，并且它们采用了相同的索引，Parallel.For 就特别适用。

参阅

3.2 节介绍如何对一批数据进行并行地聚合，包括累加和、平均值。

3.5 节介绍 PLINQ 的基础知识。

第 9 章介绍取消操作的方法。

3.2 并行聚合

问题

在并行操作结束时，需要聚合结果，包括累加和、平均值等。

解决方案

Parallel 类通过局部值（local value）的概念来实现聚合，局部值就是只在并行循环内部存在的变量。这意味着循环体中的代码可以直接访问值，不需要担心同步问题。循环中的代码使用 LocalFinally 委托来对每个局部值进行聚合。需要注意的是，localFinally 委托需要以同步的方式对存放结果的变量进行访问。下面是一个并行求累加和的例子：

```
// 注意，这不是最高效的实现方式。
// 只是举个例子，说明用锁来保护共享状态。
static int ParallelSum(IEnumerable<int> values)
{
    object mutex = new object();
    int result = 0;
    Parallel.ForEach(source: values,
        localInit: () => 0,
        body: (item, state, localValue) => localValue + item,
        localFinally: localValue =>
        {
            lock (mutex)
                result += localValue;
        });
    return result;
}
```

并行 LINQ 对聚合的支持，比 `Parallel` 类更加顺手：

```
static int ParallelSum(IEnumerable<int> values)
{
    return values.AsParallel().Sum();
}
```

好吧，那只是雕虫小技，因为 PLINQ 本身就支持很多常规操作（例如求累加和）。PLINQ 也可通过 `Aggregate` 实现通用的聚合功能：

```
static int ParallelSum(IEnumerable<int> values)
{
    return values.AsParallel().Aggregate(
        seed: 0,
        func: (sum, item) => sum + item
    );
}
```

讨论

如果程序中已经在使用 `Parallel` 类，则可使用它的聚合功能。否则，大多数情况下 PLINQ 对聚合的支持更有表现力，代码也更少。

参阅

3.5 节介绍 PLINQ 的基础知识。

3.3 并行调用

问题

需要并行调用一批方法，并且这些方法（大部分）是互相独立的。

解决方案

Parallel 类有一个简单的成员 Invoke，就可用于这种场合。下面的例子将一个数组分为两半，并且分别独立处理：

```
static void ProcessArray(double[] array)
{
    Parallel.Invoke(
        () => ProcessPartialArray(array, 0, array.Length / 2),
        () => ProcessPartialArray(array, array.Length / 2, array.Length)
    );
}

static void ProcessPartialArray(double[] array, int begin, int end)
{
    // 计算密集型的处理过程 ...
}
```

如果在运行之前都无法确定调用的数量，就可以在 Parallel.Invoke 函数中输入一个委托数组：

```
static void DoAction20Times(Action action)
{
    Action[] actions = Enumerable.Repeat(action, 20).ToArray();
    Parallel.Invoke(actions);
}
```

就像 Parallel 类的其他成员一样，Parallel.Invoke 也支持取消操作：

```
static void DoAction20Times(Action action, CancellationToken token)
{
    Action[] actions = Enumerable.Repeat(action, 20).ToArray();
    Parallel.Invoke(new ParallelOptions { CancellationToken = token }, actions);
}
```

讨论

对于简单的并行调用，Parallel.Invoke 是一个非常不错的解决方案。然而在以下两种情况中使用 Parallel.Invoke 并不是很合适：要对每一个输入的数据调用一个操作（改用 Parallel.Foreach），或者每一个操作产生了一些输出（改用并行 LINQ）。

参阅

3.1 节介绍 Parallel.ForEach，它对每个数据项调用一个操作。

3.5 节介绍 PLINQ。

3.4 动态并行

问题

并行任务的结构和数量要在运行时才能确定，这是一种更复杂的并行编程。

解决方案

任务并行库（TPL）是以 Task 类为中心构建的。Task 类的功能很强大，Parallel 类和并行 LINQ 只是为了使用方便，从而对 Task 类进行了封装。实现动态并行最简单的做法就是直接使用 Task 类。

下面的例子对二叉树的每个节点进行处理，并且该处理是很耗资源的。二叉树的结构在运行时才能确定，因此非常适合采用动态并行。Traverse 方法处理当前节点，然后创建两个子任务，每个子任务对应一个子节点（本例中，假定必须先处理父节点，然后才能处理子节点）。ProcessTree 方法启动处理过程，创建一个最高层的父任务，并等待任务完成：

```
void Traverse(Node current)
{
    DoExpensiveActionOnNode(current);
    if (current.Left != null)
    {
        Task.Factory.StartNew(() => Traverse(current.Left),
            CancellationToken.None,
            TaskCreationOptions.AttachedToParent,
            TaskScheduler.Default);
    }
    if (current.Right != null)
    {
        Task.Factory.StartNew(() => Traverse(current.Right),
            CancellationToken.None,
            TaskCreationOptions.AttachedToParent,
            TaskScheduler.Default);
    }
}

public void ProcessTree(Node root)
{
    var task = Task.Factory.StartNew(() => Traverse(root),
        CancellationToken.None,
        TaskCreationOptions.None,
        TaskScheduler.Default);
    task.Wait();
}
```

如果这些任务没有“父 / 子”关系，那可以使用任务延续（continuation）的方法，安排任务一个接着一个地运行。这里 continuation 是一个独立的任务，它在原始任务结束后运行：

```
Task task = Task.Factory.StartNew(
    () => Thread.Sleep(TimeSpan.FromSeconds(2)),
    CancellationToken.None,
    TaskCreationOptions.None,
    TaskScheduler.Default);
Task continuation = task.ContinueWith(
    t => Trace.WriteLine("Task is done"),
    CancellationToken.None,
    TaskContinuationOptions.None,
    TaskScheduler.Default);
// 对 continuation 来说，参数“t”相当于“task”
```

讨论

上面的例子使用了 `CancellationToken.None` 和 `TaskScheduler.Default`。取消令牌（cancellation token）在 9.2 节介绍，任务调度器（task scheduler）在 12.3 节介绍。最好在 `StartNew` 和 `ContinueWith` 中明确指定用 `TaskScheduler`。

在动态并行中，通常以“父 / 子”的方式对任务进行安排，但也不是必须要这么做。把每个新任务存储在线程安全的集合中，然后用 `Task.WaitAll` 等待所有任务完成，这种做法同样可行。



在并行处理中使用 `Task` 类，和在异步处理中使用完全不同。下面会详细解释。

在并发编程中，`Task` 类有两个作用：作为并行任务，或作为异步任务。并行任务可以使用阻塞的成员函数，例如 `Task.Wait`、`Task.Result`、`Task.WaitAll` 和 `Task.WaitAny`。并行任务通常也使用 `AttachedToParent` 来建立任务之间的“父 / 子”关系。并行任务的创建需要用 `Task.Run` 或者 `Task.Factory.StartNew`。

相反，异步任务应该避免使用阻塞的成员函数，而应该使用 `await`、`Task.WhenAll` 和 `Task.WhenAny`。异步任务不使用 `AttachedToParent`，但可以通过 `await` 另一个任务，建立一种隐式的“父 / 子”关系。

参阅

3.3 节介绍并行调用事先明确的一批方法。

3.5 并行LINQ

问题

需要对一批数据进行并行处理，生成另外一批数据，或者对数据进行统计。

解决方案

大部分开发者对 LINQ 比较熟悉，LINQ 可以实现在序列上”拉取“数据的运算。并行 LINQ（PLINQ）扩展了 LINQ，以支持并行处理。

PLINQ 非常适用于数据流的操作，一个数据队列作为输入，一个数据队列作为输出。下面简单的例子将序列中的每个元素都乘以 2（实际应用中，计算工作量要大得多）：

```
static IEnumerable<int> MultiplyBy2(IEnumerable<int> values)
{
    return values.AsParallel().Select(item => item * 2);
}
```

按照并行 LINQ 的默认方式，这个例子中输出数据队列的次序是不固定的。也可以指明要求保持原来的次序。下面的例子也是并行执行的，但保留了数据的原有次序：

```
static IEnumerable<int> MultiplyBy2(IEnumerable<int> values)
{
    return values.AsParallel().AsOrdered().Select(item => item * 2);
}
```

并行 LINQ 的另一个常规用途是用并行方式对数据进行聚合或汇总。下面的代码实现了并行的累加求和：

```
static int ParallelSum(IEnumerable<int> values)
{
    return values.AsParallel().Sum();
}
```

讨论

`Parallel` 类可适用于很多场合，但是在做聚合或进行数据序列的转换时，PLINQ 的代码更加简洁。有一点需要注意，相比 PLINQ，`Parallel` 类与系统中其他进程配合得更好。如果在服务器上做并行处理，这一点尤其需要考虑。

PLINQ 为各种各样的操作提供了并行的版本，包括过滤（`Where`）、投影（`Select`）以及各种聚合运算，例如 `Sum`、`Average` 和更通用的 `Aggregate`。一般来说，对常规 LINQ 的所有操作都可以通过并行方式对 PLINQ 执行。正因为如此，如果准备把已有的 LINQ 代码改为并行方式，PLINQ 是一种非常不错的选择。

参阅

3.1 节介绍利用 `Parallel` 类处理序列中每个元素的方法。

9.5 节介绍如何取消一个 PLINQ 查询。

第4章

数据流基础

TPL 数据流（dataflow）库的功能很强大，可用来创建网格（mesh）和管道（pipleline），并通过它们以异步方式发送数据。数据流的代码具有很强的“声明式编程”风格。通常要先完整地定义网格，然后才能开始处理数据，最终让网格成为一个让数据流通的体系架构。这种编程风格会让你觉得有些不适应，但一旦迈过这一步，你会发觉数据流适用于许多场合。

每个网格由各种互相链接的数据流块（block）构成。独立的块比较简单，只负责数据处理中某个单独的步骤。当块处理完它的数据后，就会把数据传递给与它链接的块。

使用 TPL 数据流之前，需要在程序中安装一个 NuGet 包：`Microsoft.Tpl.Dataflow`。各种平台对 TPL 数据流库的支持情况见表 4-1。

表4-1：支持TPL数据流的平台

平　台	数据流
.NET 4.5	✓
.NET 4.0	×
Mono iOS/Droid	✓
Windows Store	✓
Windows Phone Apps 8.1	✓
Windows Phone SL 8.0	✓
Windows Phone SL 7.1	×
Silverlight 5	×

4.1 链接数据流块

问题

创建网格时，需要把数据流块互相链接起来。

解决方案

TPL 数据流库提供的块只有一些基本的成员，很多实用的方法实际上是扩展方法。这里我们来看 LinkTo 方法：

```
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);

// 建立链接后，从 multiplyBlock 出来的数据将进入 subtractBlock。
multiplyBlock.LinkTo(subtractBlock);
```

默认情况下，链接的数据流块只传递数据，不传递完成情况（或出错信息）。如果数据流是线性的（例如管道），一般需要传递完成情况。要实现完成情况（和出错信息）的传递，可以在链接中设置 PropagateCompletion 属性：

```
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);

var options = new DataflowLinkOptions { PropagateCompletion = true };
multiplyBlock.LinkTo(subtractBlock, options);

...

// 第一个块的完成情况自动传递给第二个块。
multiplyBlock.Complete();
await subtractBlock.Completion;
```

讨论

一旦建立了链接，数据就会自动从源块传递到目标块。如果设置了 PropagateCompletion 属性，情况完成的同时也会传递数据。在管道的每个节点上，当出错的块把错误信息传递给下一块时，它会把错误信息封装进 AggregateException 对象。因此，如果传递完成情况的管道很长，错误信息就会被嵌套在很多个 AggregateException 实例中。在这种情形下，AggregateException 有几个成员（例如 Flatten）就可以进行错误处理了。

链接数据流块的方式有很多种，可以在网格中包含分叉、连接、甚至循环。不过在大多数情况下，线性的管道就足够管用了。本书主要介绍管道（此外简单地介绍一下分叉），更多的高级内容则超出了本书范围。

利用 DataflowLinkOptions 类，可以对链接设置多个不同的参数（例如前面用到的 PropagateCompletion 参数）。另外，可以在重载的 LinkTo 方法中设置断言，形成一个数据通行的过滤器。数据被过滤器拦截时也不会被删除。通过过滤器的数据会继续下一步流程，被过滤器拦截的数据也会尝试从其他链接通过，如果所有链接都无法通过，则会留在原来的块中。

参阅

4.2 节介绍如何沿着链接传递出错信息。

4.3 节介绍如何删除块之间的链接。

7.7 节介绍如何将数据流块与 Rx 可观察流链接。

4.2 传递出错信息

问题

需要处理数据流网格中发生的错误。

解决方案

如果数据流块内的委托抛出错误，这个块就进入故障状态。一旦数据流块进入故障状态，就会删除所有的数据（并停止接收新数据）。该数据流块将不会生成任何新数据。下面的代码中，第一个值引发了一个错误，第二个值被直接删除：

```
var block = new TransformBlock<int, int>(item =>
{
    if (item == 1)
        throw new InvalidOperationException("Blech.");
    return item * 2;
});
block.Post(1);
block.Post(2);
```

用 await 调用它的 Completion 属性，即可捕获数据流块的错误。Completion 属性返回一个任务，一旦数据流块执行完成，这个任务也完成。如果数据流块出错，这个任务也出错：

```
try
{
    var block = new TransformBlock<int, int>(item =>
    {
        if (item == 1)
            throw new InvalidOperationException("Blech.");
        return item * 2;
```

```
    });
    block.Post(1);
    await block.Completion;
}
catch (InvalidOperationException)
{
    // 这里捕获异常。
}
```

如果用 PropagateCompletion 这个参数传递完成情况，错误信息也会被传递。只不过这个异常是被封装在 AggregateException 类中传递给下一个块。下面的例子中，程序在管道的末尾捕获到了异常。这说明，如果异常是从前面的块传来的，程序就会捕获到 AggregateException：

```
try
{
    var multiplyBlock = new TransformBlock<int, int>(item =>
    {
        if (item == 1)
            throw new InvalidOperationException("Blech.");
        return item * 2;
    });
    var subtractBlock = new TransformBlock<int, int>(item => item - 2);
    multiplyBlock.LinkTo(subtractBlock,
        new DataflowLinkOptions { PropagateCompletion = true });
    multiplyBlock.Post(1);
    await subtractBlock.Completion;
}
catch (AggregateException)
{
    // 这里捕获异常。
}
```

数据流块收到传过来的出错信息后，即使它已经被封装在 AggregateException，仍会用 AggregateException 进行封装。如果在管道的前面部分发生错误，经过了多个链接后才被发现，这个原始错误就会被 AggregateException 封装很多层。这时用 AggregateException.Flatten 方法可以简化错误处理过程。

讨论

思考一下这个问题，在构建了网格（或管道）后怎么处理错误。对于最简单的情况，最好是把错误传递下去，等到最后再作一次性处理。对于更复杂的网格，在数据流完成后需要检查每一个数据流块。

参阅

4.1 节介绍如何链接数据流块。

4.3 节介绍如何断开数据流块的链接。

4.3 断开链接

问题

要在处理的过程中修改数据流结构。这是一种高级应用，很少会用到。

解决方案

可以随时对数据流块建立链接或断开链接。数据在网格中的自由传递，不会受此影响。建立或断开链接时，线程都是完全安全的。

在创建数据流块之间的链接时，保留 `LinkTo` 方法返回的 `IDisposable` 接口。想断开它们的链接时，只需释放该接口：

```
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);

IDisposable link = multiplyBlock.LinkTo(subtractBlock);
multiplyBlock.Post(1);
multiplyBlock.Post(2);

// 断开数据流块的链接。
// 前面的代码中，数据可能已经通过链接传递过去，也可能还没有。
// 在实际应用中，考虑使用代码块，而不是调用 Dispose。
link.Dispose();
```

讨论

除非能保证链接是空闲的，否则在断开数据流块的链接时就会出现竞态条件（race condition）。但是，通常不需要担心这类竞态条件。数据要么在链接断开之前就已经传递到下一块，要么就永远不会传递。这些竞态条件不会重复出现数据，也不会丢失数据。

断开链接是一个高级应用，但它仍能用于一些场合。举个例子，在链接建立后是无法修改过滤器的，要修改一个已有链接的过滤器，必须先断开旧链接，然后用新的过滤器建立新链接（可以把 `DataflowLinkOptions.Append` 设为 `false`）。另外，要暂停数据流网格运行的话，可断开一个关键链接。

参阅

4.1 节介绍如何在数据流块之间建立链接。

4.4 限制流量

问题

需要在数据流网格中进行分叉，并且希望数据流量能在各分支之间平衡。

解决方案

默认情况下，数据流块生成输出的数据后，会检查每个链接（按照创建的次序），逐个地尝试通过链接传递数据。同样，默认情况下，每个数据流块会维护一个输入缓冲区，在处理数据之前接收任意数量的数据。

有分叉时，一个源块链接了两个目标块，上述做法就会产生问题：第一个目标块会不停地缓冲数据，第二个目标块就永远没有机会得到数据。这个问题的解决办法是使用数据流块的 BoundedCapacity 属性，来限制目标块的流量（throttling）。BoundedCapacity 的默认设置是 DataflowBlockOptions.Unbounded，这会导致第一个目标块在还来不及处理数据时就得对所有数据进行缓冲了。

BoundedCapacity 可以是大于 0 的任何数值（当然也可以是 DataflowBlockOptions.Unbounded）。只要目标块来得及处理来自源块的数据，将这个参数设为 1 就足够了：

```
var sourceBlock = new BufferBlock<int>();
var options = new DataflowBlockOptions { BoundedCapacity = 1 };
var targetBlockA = new BufferBlock<int>(options);
var targetBlockB = new BufferBlock<int>(options);

sourceBlock.LinkTo(targetBlockA);
sourceBlock.LinkTo(targetBlockB);
```

讨论

限流可用于分叉的负载平衡，但也可用在任何限流行为中。例如，在用 I/O 操作的数据填充数据流网格时，可以设置数据流块的 BoundedCapacity 属性。这样，在网格来不及处理数据时，就不会读取过多的 I/O 数据，网格也不会缓存所有数据。

参阅

4.1 节介绍建立数据流块之间的链接。

4.5 数据流块的并行处理

问题

想对数据流网格进行并行处理。

解决方案

默认情况下每个数据流块是互相独立的。将两个数据流块链接起来后，它们也是独立运行的。因此每个数据流网格本身就有并行特性。

如果想更进一步，例如某个特定的数据流块的计算量特别大，那就可以设置 `MaxDegreeOfParallelism` 参数，使数据流块在处理输入的数据时采用并行的方式。`MaxDegreeOfParallelism` 的默认值是 1，因此每个数据流块同时只能处理一块数据。

`MaxDegreeOfParallelism` 可以设为 `DataflowBlockOptions.Unbounded` 或任何大于 0 的值。下面的例子允许任意数量的任务，来同时对数据进行倍增：

```
var multiplyBlock = new TransformBlock<int, int>(
    item => item * 2,
    new ExecutionDataflowBlockOptions
    {
        MaxDegreeOfParallelism = DataflowBlockOptions.Unbounded
    }
);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);
multiplyBlock.LinkTo(subtractBlock);
```

讨论

利用 `MaxDegreeOfParallelism` 参数，就可以很容易地在数据流块中实现并行处理。而真正的难点，在于找出哪些数据流块需要并行处理，有一个办法是在调试时暂停数据流的运行，在调试器中查看等待的数据项的数量（就是还没有被数据流块处理的数据项）。如果等待的数据项很多，就表明需要进行重构或并行化处理。

在数据流块进行异步处理时，`MaxDegreeOfParallelism` 参数也会发挥作用。这时，`MaxDegreeOfParallelism` 参数代表的是并发的层次，即一定数量的槽（slot）。在数据流块开始处理数据项之际，每个数据项就会占用一个槽。只有当整个异步处理过程完成后，才会释放槽。

参阅

4.1 节介绍链接数据流块。

4.6　创建自定义数据流块

问题

希望一些可重用的程序逻辑在自定义数据流块中使用。这有助于创建更大的、包含复杂逻辑的数据流快。

解决方案

通过使用 Encapsulate 方法，可以取出数据流网格中任何具有单一输入块和输出块的部分。Encapsulate 方法会利用这两个端点，创建一个单独的数据流块。开发者得自己负责端点之间数据的传递以及完成情况。下面的代码利用两个数据流块创建了一个自定义数据流块，并实现了数据和完成情况的传递：

```
IPropagatorBlock<int, int> CreateMyCustomBlock()
{
    var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
    var addBlock = new TransformBlock<int, int>(item => item + 2);
    var divideBlock = new TransformBlock<int, int>(item => item / 2);

    var flowCompletion = new DataflowLinkOptions { PropagateCompletion = true };
    multiplyBlock.LinkTo(addBlock, flowCompletion);
    addBlock.LinkTo(divideBlock, flowCompletion);

    return DataflowBlock.Encapsulate(multiplyBlock, divideBlock);
}
```

讨论

在把一个网格封装成一个自定义数据流块时，得考虑一下对外提供什么类型的参数，考虑每个块参数应该怎样传递进内部的网格（或不传递）。在很多情况下，有些块参数是不适合的，或者是没有意义的。基于这个原因，创建自定义数据流块时，通常得自行定义参数，而不是沿用 DataflowBlockOptions 参数。

DataflowBlock.Encapsulate 只会封装只有一个输入块和一个输出块的网格。如果一个可重用的网格带有多个输入或输出，就应该把它封装进一个自定义对象，并以属性的形式对外暴露出这些输入和输出，输入的属性类型是 ITargetBlock<T>，输出的属性类型是 IReceivableSourceBlock<T>。

前面的例子都采用 Encapsulate 来创建自定义数据流块。开发者也可以自行实现数据流的接口，但技术难度很大。创建自定义数据流块的高级技术，可参阅微软公司发布的有关文章。

参阅

4.1 节介绍了链接数据流块。

4.2 节介绍了通过块的链接传递出错信息。

第5章

Rx基础

LINQ是对序列数据进行查询的一系列语言功能。内置的LINQ to Objects（基于`IEnumerable<T>`）和LINQ to Entities（基于`IQueryableT>`）是两个最常用的`LINQ`提供者。另外还有很多提供者，并且大多数都采用相同的基本架构。查询是延后执行（lazily evaluated）的，只有在需要时才会从序列中获取数据。从概念上讲，这是一种拉取模式。在查询过程中数据项是被逐个拉取出来的。

Reactive Extensions（Rx）把事件看作是依次到达的数据序列。因此，将Rx认作是LINQ to events（基于`IObservable<T>`）也是可以的，它与其他LINQ提供者的主要区别在于，Rx采用“推送”模式。就是说，Rx的查询规定了在事件到达时程序该如何响应。Rx在LINQ的基础上构建，增加了一些功能强大的操作符，作为扩展方法。

本章介绍一些更常用的Rx操作。需要注意的是，所有的LINQ操作都可以在Rx中使用。从概念上看，过滤（`Where`）、投影（`Select`）等简单操作，和其他LINQ提供者的操作是一样的。本章不介绍那些常见的LINQ操作，而将重点放在Rx在LINQ基础上增加的新功能，尤其是与时间有关的功能。

要使用Rx，需要在应用中安装一个NuGet包`Rx-Main`。支持Reactive Extensions的平台非常丰富（参见表5-1）。

表5-1：支持Reactive Extensions的平台

平　台	Rx支持情况
.NET 4.5	✓
.NET 4.0	✓

（续）

平　　台	Rx支持情况
Mono iOS/Droid	√
Windows Store	√
Windows Phone Apps 8.1	√
Windows Phone SL 8.0	√
Windows Phone SL 7.1	√
Silverlight 5	√

5.1　转换.NET事件

问题

把一个事件作为 Rx 输入流，每次事件发生时通过 OnNext 生成数据。

解决方案

Observable 类定义了一些事件转换器。大部分 .NET 框架事件与 FromEventPattern 兼容，对于不遵循通用模式的事件，需要改用 FromEvent。

FromEventPattern 最适合使用委托类型为 EventHandler<T> 的事件。很多较新框架类的事件都采用了这种委托类型。例如，Progress<T> 类定义了事件 ProgressChanged，这个事件的委托类型就是 EventHandler<T>，因此，它就很容易被封装到 FromEventPattern：

```
var progress = new Progress<int>();
var progressReports = Observable.FromEventPattern<int>(
    handler => progress.ProgressChanged += handler,
    handler => progress.ProgressChanged -= handler);
progressReports.Subscribe(data => Trace.WriteLine("OnNext:" + data.EventArgs));
```

请注意，data.EventArgs 是强类型的 int。FromEventPattern 的类型参数（上例中为 int）与 EventHandler<T> 的 T 相同。Rx 用 FromEventPattern 中的两个 Lambda 参数来实现订阅和退订事件。

较新的 UI 框架采用 EventHandler<T>，可以很方便地应用在 FromEventPattern 中。但是有些较旧的类常为每个事件定义不同的委托类型。这些事件也能在 FromEventPattern 中使用，但需要做一些额外的工作。例如，System.Timers.Timer 类有一个事件 Elapsed，它的类型是 ElapsedEventHandler。对此旧类事件，可以用下面的方法封装进 FromEventPattern：

```
var timer = new System.Timers.Timer(interval: 1000) { Enabled = true };
var ticks = Observable.FromEventPattern<ElapsedEventHandler, ElapsedEventArgs>(
    handler => (s, a) => handler(s, a),
    handler => timer.Elapsed += handler,
```

```
        handler => timer.Elapsed -= handler);
    ticks.Subscribe(data => Trace.WriteLine("OnNext: " + data.EventArgs.SignalTime));
```

注意，data.EventArgs 仍然是强类型的。现在 FromEventPattern 的类型参数是对应的事件处理程序和 EventArgs 的派生类。FromEventPattern 的第一个 Lambda 参数是一个转换器，它将 EventHandler<ElapsedEventArgs> 转换成 ElapsedEventHandler。除了传递事件，这个转换器不应该做其他处理。

上面代码的语法明显有些别扭。另一个方法是使用反射机制：

```
    var timer = new System.Timers.Timer(interval: 1000) { Enabled = true };
    var ticks = Observable.FromEventPattern(timer, "Elapsed");
    ticks.Subscribe(data => Trace.WriteLine("OnNext: "
        + ((ElapsedEventArgs)data.EventArgs).SignalTime));
```

采用这种方法后，调用 FromEventPattern 就简单多了。但是这种方法也有缺点：出现了一个怪异的字符串（"Elapsed"），并且消息的使用者不是强类型了。就是说，这时 data.EventArgs 是 object 类型，需要人为地转换成 ElapsedEventArgs。

讨论

事件是 Rx 流数据的主要来源。本节介绍如何封装遵循标准模式的事件（标准事件模式：第一个参数是事件发送者，第二个参数是事件的类型参数）。对于不标准的事件类型，可以用重载 Observable.FromEvent 的办法，把事件封装进 Observable 对象。

把事件封装进 Observable 对象后，每次引发该事件都会调用 OnNext。在处理 AsyncCompletedEventArgs 时会发生令人奇怪的现象，所有的异常信息都是通过数据形式传递的（OnNext），而不是通过错误传递（OnError）。看一个封装 WebClient.DownloadStringCompleted 的例子：

```
    var client = new WebClient();
    var downloadedStrings = Observable.FromEventPattern(client,
        "DownloadStringCompleted");
    downloadedStrings.Subscribe(
        data =>
        {
            var eventArgs = (DownloadStringCompletedEventArgs)data.EventArgs;
            if (eventArgs.Error != null)
                Trace.WriteLine("OnNext: (Error) " + eventArgs.Error);
            else
                Trace.WriteLine("OnNext: " + eventArgs.Result);
        },
        ex => Trace.WriteLine("OnError: " + ex.ToString()),
        () => Trace.WriteLine("OnCompleted"));
    client.DownloadStringAsync(new Uri("http://invalid.example.com/"));
```

WebClient.DownloadStringAsync 出错并结束时，引发带有异常 AsyncCompletedEventArgs.Error

的事件。可惜 Rx 会把这作为一个数据事件，因此这个程序的结果是显示“`OnNext:(Error)`”，而不是“`OnError:`”。

有些事件的订阅和退订必须在特定的上下文中进行。例如，很多 UI 控件的事件必须在 UI 线程中订阅。Rx 提供了一个操作符 `SubscribeOn`，可以控制订阅和退订的上下文。大多数情况下没必要使用这个操作符，因为基于 UI 的事件订阅通常就是在 UI 线程中进行的。

参阅

5.2 节介绍如何修改引发事件的上下文。

5.4 节介绍如何对事件限流，以免订阅者因事件太多而崩溃。

5.2 发通知给上下文

问题

Rx 尽量做到了线程不可知（thread agnostic）。因此它会在任意一个活动线程中发出通知（例如 `OnNext`）。

但是我们通常希望通知只发给特定的上下文。例如 UI 元素只能被它所属的 UI 线程控制，因此，如果要根据 Rx 的通知来修改 UI，就应该把通知“转移”到 UI 线程。

解决方案

Rx 提供了 `ObserveOn` 操作符，用来把通知转移到其他线程调度器。

看下面的例子，使用 `Interval`，每秒钟产生一个 `OnNext` 通知：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Trace.WriteLine("UI thread is " + Environment.CurrentManagedThreadId);
    Observable.Interval(TimeSpan.FromSeconds(1))
        .Subscribe(x => Trace.WriteLine("Interval " + x + " on thread " +
            Environment.CurrentManagedThreadId));
}
```

用我的电脑测试，显示结果为：

```
UI thread is 9
Interval 0 on thread 10
Interval 1 on thread 10
Interval 2 on thread 11
Interval 3 on thread 11
Interval 4 on thread 10
```

```
Interval 5 on thread 11
Interval 6 on thread 11
```

因为 Interval 基于一个定时器（没有指定的线程），通知会在线程池线程中引发，而不是在 UI 线程中。要更新 UI 元素，可以通过 ObserveOn 输送通知，并传递一个代表 UI 线程的同步上下文：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var uiContext = SynchronizationContext.Current;
    Trace.WriteLine("UI thread is " + Environment.CurrentManagedThreadId);
    Observable.Interval(TimeSpan.FromSeconds(1))
        .ObserveOn(uiContext)
        .Subscribe(x => Trace.WriteLine("Interval " + x + " on thread " +
            Environment.CurrentManagedThreadId));
}
```

ObserveOn 的另一个常用功能是可以在必要时离开 UI 线程。假设有这样的情况：鼠标一移动，就意味着需要进行一些 CPU 密集型的计算。默认情况下，所有的鼠标移动事件都发生在 UI 线程，因此可以使用 ObserveOn 把通知移动到一个线程池线程，在那里进行计算，然后再把表示结果的通知返回给 UI 线程：


```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var uiContext = SynchronizationContext.Current;
    Trace.WriteLine("UI thread is " + Environment.CurrentManagedThreadId);
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Select(evt => evt.EventArgs.GetPosition(this))
        .ObserveOn(Scheduler.Default)
        .Select(position =>
        {
            // 复杂的计算过程。
            Thread.Sleep(100);
            var result = position.X + position.Y;
            Trace.WriteLine("Calculated result " + result + " on thread " +
                Environment.CurrentManagedThreadId);
            return result;
        })
        .ObserveOn(uiContext)
        .Subscribe(x => Trace.WriteLine("Result " + x + " on thread " +
            Environment.CurrentManagedThreadId));
}
```

运行这段代码的话，就会发现计算过程是在线程池线程中进行的，计算结果在 UI 线程中显示。另外，还会发现计算和结果会滞后于输入，形成等待的队列，这种现象出现的原因在于，比起 100 秒 1 次的计算，鼠标移动的更新频率更高。Rx 中有几种技术可以处理这种情况，其中一个常用方法是对输入流速进行限制，具体会在 5.4 节介绍。

讨论

实际上，ObserveOn 是把通知转移到一个 Rx 调度器上了。本节介绍了默认调度器（即线程池）和一种创建 UI 调度器的方法。ObserveOn 最常用的功能是移到或移出 UI 线程，但调度器也能用于别的场合。6.6 节介绍高级测试时，将再次关注调度器。

参阅

5.1 节介绍如何利用事件创建序列。

5.4 节介绍如何限制事件流的流速。

6.6 节介绍测试 Rx 代码的特殊流程。

5.3 用窗口和缓冲对事件分组

问题

有一系列事件，需要在它们到达时进行分组。举个例子，需要对一些成对的输入作出响应。第二个例子，需要在 2 秒钟的窗口期内，对所有输入进行响应。

解决方案

Rx 提供了两个对到达的序列进行分组的操作：Buffer 和 Window。Buffer 会留住到达的事件，直到收完一组事件，然后会把这一组事件以一个集合的形式一次性地转送过去。Window 会在逻辑上对到达的事件进行分组，但会在每个事件到达时立即传递过去。Buffer 的返回类型是 IObservable<IList<T>>（由若干个集合组成的事件流）；Window 的返回类型是 IObservable<IObservable<T>>（由若干个事件流组成的事件流）。

下面的例子使用 Interval，每秒创建 1 个 OnNext 通知，然后每 2 个通知做一次缓冲：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.Interval(TimeSpan.FromSeconds(1))
        .Buffer(2)
        .Subscribe(x => Trace.WriteLine(
            DateTime.Now.Second + ": Got " + x[0] + " and " + x[1]));
}
```

用我的电脑测试，每 2 秒产生 2 个输出：

```
13: Got 0 and 1
15: Got 2 and 3
17: Got 4 and 5
```

```
19: Got 6 and 7
21: Got 8 and 9
```

下面的例子有些类似，使用 Window 创建一些事件组，每组包含 2 个事件：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.Interval(TimeSpan.FromSeconds(1))
        .Window(2)
        .Subscribe(group =>
        {
            Trace.WriteLine(DateTime.Now.Second + ": Starting new group");
            group.Subscribe(
                x => Trace.WriteLine(DateTime.Now.Second + ": Saw " + x),
                () => Trace.WriteLine(DateTime.Now.Second + ": Ending group"));
        });
}
```

用我的电脑测试，输出的结果就是这样：

```
17: Starting new group
18: Saw 0
19: Saw 1
19: Ending group
19: Starting new group
20: Saw 2
21: Saw 3
21: Ending group
21: Starting new group
22: Saw 4
23: Saw 5
23: Ending group
23: Starting new group
```

这几个例子说明了 Buffer 和 Window 的区别。Buffer 等待组内的所有事件，然后把所有事件作为一个集合发布。Window 用同样的方法进行分组，但它是在每个事件到达时就发布。

Buffer 和 Window 都可以使用时间段作为参数。在下面的例子中，所有的鼠标移动事件被收集进窗口，每秒一个窗口：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Buffer(TimeSpan.FromSeconds(1))
        .Subscribe(x => Trace.WriteLine(
            DateTime.Now.Second + ": Saw " + x.Count + " items."));
}
```

输出的结果依赖于怎么移动鼠标，类似于这样：

```
49: Saw 93 items.
50: Saw 98 items.
51: Saw 39 items.
52: Saw 0 items.
53: Saw 4 items.
54: Saw 0 items.
55: Saw 58 items.
```

讨论

Buffer 和 Window 可用来抑制输入信息，并把输入塑造成我们想要的样子。另一个实用技术是限流（throttling），将在 5.4 节介绍。

Buffer 和 Windows 都有其他重载，可用在更高级的场合。参数为 skip 和 timeShift 的重载能创建互相重合的组，还可跳过组之间的元素。还有一些重载可使用委托，可对组的边界进行动态定义。

参阅

5.1 节介绍如何利用事件创建序列。

5.4 节介绍对事件流进行限流。

5.4 用限流和抽样抑制事件流

问题

有时事件来得太快，这是编写响应式代码时经常碰到的问题。一个速度太快的事件流可导致程序的处理过程崩溃。

解决方案

Rx 专门提供了几个操作符，用来对付大量涌现的事件数据。Throttle 和 Sample 这两个操作符提供了两种不同方法来抑制快速涌来的输入事件。

Throttle 建立了一个超时窗口，超时期限可以设置。当一个事件到达时，它就重新开始计时。当超时期限到达时，它就把窗口内到达的最后一个事件发布出去。

下面的例子也是监视鼠标移动，但使用了 Throttle，在鼠标保持静止 1 秒后才报告最近一条移动事件。

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
```

```
                handler => (s, a) => handler(s, a),
                handler => MouseMove += handler,
                handler => MouseMove -= handler)
            .Select(x => x.EventArgs.GetPosition(this))
            .Throttle(TimeSpan.FromSeconds(1))
            .Subscribe(x => Trace.WriteLine(
                DateTime.Now.Second + ": Saw " + (x.X + x.Y)));
}
```

输出结果依赖于鼠标的实际动作，我的测试结果是这样：

```
47: Saw 139
49: Saw 137
51: Saw 424
56: Saw 226
```

Throttle 常用于类似“文本框自动填充”这样的场合，用户在文本框中输入文字，当他停止输入时，才需要进行真正的检索。

为抑制快速运动的事件序列，Sample 操作符使用了另一种方法。Sample 建立了一个有规律的超时时间段，每个时间段结束时，它就发布该时间段内最后的一条数据。如果这个时间段没有数据，就不发布。

下面的例子捕获鼠标移动，每隔一秒采样一次。与 Throttle 不同，使用 Sample 的例子中，不需要让鼠标静止一段时间，就可要看到结果。

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
                handler => (s, a) => handler(s, a),
                handler => MouseMove += handler,
                handler => MouseMove -= handler)
            .Select(x => x.EventArgs.GetPosition(this))
            .Sample(TimeSpan.FromSeconds(1))
            .Subscribe(x => Trace.WriteLine(
                DateTime.Now.Second + ": Saw " + (x.X + x.Y)));
}
```

我先让鼠标静止几秒钟，然后连续移动，得到了下面的输出结果：

```
12: Saw 311
17: Saw 254
18: Saw 269
19: Saw 342
20: Saw 224
21: Saw 277
```

讨论

对于快速涌来的输入，限流和抽样是很重要的两种工具。别忘了还有一个过滤输入的简单

方法，就是采用标准 LINQ 的 Where 操作符。可以这样说，Throttle 和 Sample 操作符与 Where 基本差不多，唯一的区别是 Throttle、Sample 根据时间段过滤，而 Where 根据事件的数据过滤。在抑制快速涌来的输入流时，这三种操作符提供了三种不同的方法。

参阅

5.1 节介绍如何创建事件序列。

5.2 节介绍如何修改引发事件的上下文。

5.5 超时

问题

我们希望事件能在预定的时间内到达，即使事件不到达，也要确保程序能及时进行响应。通常此类事件是单一的异步操作（例如，等待 Web 服务请求的响应）。

解决方案

Timeout 操作符在输入流上建立一个可调节的超时窗口。一旦新的事件到达，就重置超时窗口。如果超过期限后事件仍没到达，Timeout 操作符就结束流，并产生一个包含 TimeoutException 的 OnError 通知。

下面的代码向一个域名发出 Web 请求，并使用 1 秒作为超时值：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var client = new HttpClient();
    client.GetStringAsync("http://www.example.com/").ToObservable()
        .Timeout(TimeSpan.FromSeconds(1))
        .Subscribe(
            x => Trace.WriteLine(DateTime.Now.Second + ": Saw " + x.Length),
            ex => Trace.WriteLine(ex));
}
```

Timeout 非常适用于异步操作，例如 Web 请求，但它也能用于任何事件流。下面的例子在监视鼠标移动时使用 Timeout，使用起来更加简单：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Select(x => x.EventArgs.GetPosition(this))
```

```
            .Timeout(TimeSpan.FromSeconds(1))
            .Subscribe(
                x => Trace.WriteLine(DateTime.Now.Second + ": Saw " + (x.X + x.Y)),
                ex => Trace.WriteLine(ex));
    }
```

我移动了一下鼠标，然后停止 1 秒，得到如下结果：

```
16: Saw 180
16: Saw 178
16: Saw 177
16: Saw 176
System.TimeoutException: The operation has timed out.
```

值得注意的是，一旦向 `OnError` 发送 `TimeoutException`，整个事件流就结束了，不会继续传来鼠标移动事件。为了阻止这种情况出现，`Timeout` 操作符具有重载方式，在超时发生时用另一个流来替代，而不是抛出异常并结束流。

下面的例子，在超时之前观察鼠标移动，超时发生后进行切换，观察鼠标点击：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var clicks = Observable.FromEventPattern
        <MouseButtonEventHandler, MouseButtonEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseDown += handler,
            handler => MouseDown -= handler)
        .Select(x => x.EventArgs.GetPosition(this));

    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Select(x => x.EventArgs.GetPosition(this))
        .Timeout(TimeSpan.FromSeconds(1), clicks)
        .Subscribe(
            x => Trace.WriteLine(
                DateTime.Now.Second + ": Saw " + x.X + "," + x.Y),
            ex => Trace.WriteLine(ex));
}
```

我先移动一下鼠标，停止 1 秒，然后在两个不同的位置点击。下面的输出表明，超时发生前鼠标移动事件在进行快速移动，超时后变成两个鼠标点击事件：

```
49: Saw 95,39
49: Saw 94,39
49: Saw 94,38
49: Saw 94,37
53: Saw 130,141
55: Saw 469,4
```

讨论

`Timeout` 操作符对优秀的程序来说是十分必要的，因为我们总是希望程序能及时响应，即使外部环境不理想。它可用于任何事件流，尤其是在异步操作时。需要注意，此时内部的操作并没有真正取消，操作将继续执行，直到成功或失败。

参阅

5.1 节介绍如何利用事件创建序列。

7.6 节介绍如何把异步代码封装成 Observable 对象事件流。

9.6 节介绍收到 `CancellationToken` 时如何从序列中退订。

9.3 节介绍用 `CancellationToken` 来实现超时功能。

第6章

测试技巧

测试是保证软件质量必不可少的环节。近年来，提倡单元测试的人越来越多，到处都能听到有关单元测试的讨论。有人提倡测试驱动型的开发模式，以保证软件测试和开发同步进行、同时完成。大家都知道单元测试在保证代码质量和整个开发过程中的作用，然而大多数开发人员直到今天都没有真正编写过单元测试。

我建议大家至少写一些单元测试，首先从自己觉得最没信心的代码开始。根据我个人的经验，单元测试主要有两大好处。

(1) 更好地理解代码。你是否遇到过这种情况：你了解程序的某个部分能正常运行，却对它的实现原理一无所知。当软件出现了令你不可思议的错误时，这种疑问常常占据你的内心深处。要理解那些特别“难”的代码的内部机理，编写单元测试就是一个很好的办法。编写描述代码行为的单元测试之后，就不会觉得这部分代码神秘了。编写一批单元测试后，最终就能搞清那些代码的行为，以及它们和其他代码之间的依赖关系。

(2) 修改代码时更有把握。迟早会有那么一天，你会因为有功能需求而必须修改那些“恐怖”的代码，你将无法继续假装它不存在。（我了解那种感觉。我经历过！）最好提前做好准备：在此类需求到来之前，为那些恐怖的代码编写单元测试。提前准备，以免以后麻烦。如果你的单元测试是完整的，你就相当于有了一个早期预警系统，如果修改后的代码影响到已有功能时，它就会立即发出警告。

不管是你自己还是其他人的代码，都能获得上述好处。我敢肯定单元测试还能带来其他好处。单元测试能减少错误出现的频率吗？很有可能。单元测试能减少项目的整体时间吗？有可能。但是我在上面列出的几条好处是肯定会有的。我每次编写单元测试时都能感受到。

因此，我强烈推荐单元测试。

本章的内容全部是关于测试的。很多开发人员（甚至包括经常编写单元测试的人）都逃避并发代码的单元测试，因为他们总觉得非常难。然而本章的内容将会告诉大家，并发代码的单元测试并没有想象中那么难。现在的语言功能和开发库，例如 async 和 Rx，在测试的方便性方面做了很多考虑，并且确实能体现出这点。我建议大家使用本章的方法编写单元测试，尤其是并发编程的新手（就是认为新并发代码“很难”或“可怕”的人）。

6.1 async方法的单元测试

问题

需要对 async 方法进行单元测试。

解决方案

现在大多数单元测试框架都支持 async Task 类型的单元测试，包括 MSTest、NUnit、xUnit。从 Visual Studio 2012 开始，MSTest 才支持 async Task 类型的单元测试，因此需要将老版本升级到最新版本。

下面是一个 async 类型 MSTest 单元测试的例子：

```
[TestMethod]
public async Task MyMethodAsync_ReturnsFalse()
{
    var objectUnderTest = ...;
    bool result = await objectUnderTest.MyMethodAsync();
    Assert.IsFalse(result);
}
```

单元测试框架检测到方法的返回类型是 Task，会自动 加上 await 等待任务完成，然后将测试结果标记为“成功”或“失败”。

如果单元测试框架不支持 async Task 类型的单元测试，就需要做一些额外的修改才能等待异步操作。其中一种做法是使用 Task.Wait，并在有错误时拆开 AggregateException 对象。我的建议是使用 NuGet 包 Nito.AsyncEx 中的 AsyncContext 类：

```
[TestMethod]
public void MyMethodAsync_ReturnsFalse()
{
    AsyncContext.Run(async () =>
    {
        var objectUnderTest = ...;
        bool result = await objectUnderTest.MyMethodAsync();
        Assert.IsFalse(result);
```

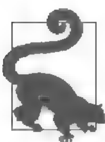
```
        });
    }
```

AsyncContext.Run 会等待所有异步方法完成。

讨论

模拟（mocking）异步方法间的依赖关系，虽说它给人的第一感觉是有点别扭，但至少可以测试某些方法如何响应同步成功（用 Task.FromResult 模拟）、同步出错（用 TaskCompletionSource<T> 模拟）以及异步成功（用 Task.Yield 模拟，并返回一个值），并且它在做这些测试时，是一个很好的办法。

跟同步代码相比，在测试异步代码时会出现更多的死锁和竞态条件。我发现，对每个测试进行超时设置很有用。在 Visual Studio 中，可以在解决方案中加一个测试设置文件，用来对每个测试设置独立的超时参数。这个参数的默认值是很大的，我通常将每一个测试的超时参数设成 2 秒。



AsyncContext 类在 NuGet 包的 Nito.AsyncEx 中。

参阅

6.2 节介绍对预计失败的异步方法进行单元测试。

6.2 预计失败的async方法的单元测试

问题

需要编写一个单元测试，用来检查 async Task 方法的一个特定错误。

解决方案

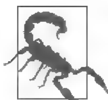
对于桌面程序或服务器程序，MSTest 就可以用常规的 ExpectedExceptionAttribute 进行错误测试：

```
// 不推荐用这种方法，原因在后面。
[TestMethod]
[ExpectedException(typeof(DivideByZeroException))]
public async Task Divide_WhenDenominatorIsZero_ThrowsDivideByZero()
{
    await MyClass.DivideAsync(4, 0);
}
```

但是这种方法并不是最好的。一方面，Windows 应用商店并没有 ExpectedException 支持单元测试。另一个本质的原因是 ExpectedException 的设计非常糟糕。单元测试中调用的任何方法都可以抛出这个异常。更好的设计是检查抛出异常的那段代码，而不是检查整个单元测试。

微软公司已经在向这个方向努力了，在 Windows 应用商店单元测试中去掉了 ExpectedException，改成用 Assert.ThrowsException<TException>，使用方法如下：

```
[TestMethod]
public async Task Divide_WhenDenominatorIsZero_ThrowsDivideByZero()
{
    await Assert.ThrowsException<DivideByZeroException>(async () =>
    {
        await MyClass.DivideAsync(4, 0);
    });
}
```



千万别忘了对 ThrowsException 返回的 Task 使用 await。这样才可以传递出所有监测到的出错信息。如果忘记使用 await 并且忽视了编译器的警告，那么不管被测试的方法是否真的正确，单元测试就会一直显示测试成功且不给任何提示。

可惜，微软只在 Windows 应用商店单元测试项目中加入了 ThrowsException，到目前为止其他几种单元测试框架并没有与 ThrowsException 等效的、兼容 async 的方法。这时我们可以自行创建这样的方法：

```
/// <summary>
/// 确保一个异步委托抛出异常。
/// </summary>
/// <typeparam name="TException">
/// 所预计异常的类型。
/// </typeparam>
/// <param name="action">被测试的异步委托</param>
/// <param name="allowDerivedTypes">
/// 是否接受派生的类。
/// </param>
public static async Task ThrowsExceptionAsync<TException>(Func<Task> action,
    bool allowDerivedTypes = true)
{
    try
    {
        await action();
        Assert.Fail("Delegate did not throw expected exception " +
            typeof(TException).Name + ".");
    }
    catch (Exception ex)
    {
        if (allowDerivedTypes && !(ex is TException))
```

```
                Assert.Fail("Delegate threw exception of type " + ex.GetType().Name +
                    ", but " + typeof(TException).Name +
                    " or a derived type was expected.");
            if (!allowDerivedTypes && ex.GetType() != typeof(TException))
                Assert.Fail("Delegate threw exception of type " + ex.GetType().Name +
                    ", but " + typeof(TException).Name + " was expected.");
        }
    }
```

调用这个方法跟 Windows 应用商店的 MSTest 方法 `Assert.ThrowsException<TException>` 一样。千万别忘记对返回值使用 `await`！

讨论

对错误处理进行测试，与测试正确的场景一样重要。甚至有人认为前者更重要，因为正确场景是每个人在软件发布前就试过的。如果软件的运行情况很怪异，可能是因为出现了以前没预料到的错误情形。

不过，我建议大家不要使用 `ExpectedException`。它更适用于测试某个特定点抛出的异常，而不是整个测试过程中随时会抛出的异常。不用 `ExpectedException` 的话，就可改用 `ThrowsException`（或者单元测试框架中类似的方法），或者使用它的另一种实现 `ThrowsExceptionAsync`。

参阅

6.1 节介绍异步方法单元测试的基础知识。

6.3 async void方法的单元测试

问题

需要对一个 `asnyc void` 类型的方法做单元测试。

解决方案

停。

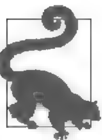
要尽最大可能避免这个问题，而不是去解决它。只要有可能把 `async void` 方法改成 `async Task`，那就得改。

如果一个方法必须采用 `async void`（例如为满足某个接口方法的特征），那可考虑编写两个方法：一个包含所有逻辑的 `async Task` 方法和一个 `async void` 方法。这个 `async void` 方法只是做一个简单封装，即调用 `async Task` 方法，并用 `await` 等待结果。这样，`async void` 方法可满足格式要求，而 `async Task` 方法（包含所有逻辑）可用于测试。

如果真的不可能修改这个方法，并且确实必须对一个 async void 方法做单元测试，可试试这个方法，使用 Nito.AsyncEx 类库的 AsyncContext 类：

```
// 不推荐用这种方法，原因见前面。
[TestMethod]
public void MyMethodAsync_DoesNotThrow()
{
    AsyncContext.Run(() =>
    {
        var objectUnderTest = ...;
        objectUnderTest.MyMethodAsync();
    });
}
```

这个 AsyncContext 类会等待所有异步操作完成（包括 async void 方法），再将异常传递出去。



AsyncContext 在 NuGet 包 Nito.AsyncEx 中。

讨论

在 async 代码中，关键准则之一就是避免使用 async void。我非常建议大家在对 async void 方法做单元测试时进行代码重构，而不是使用 AsyncContext。

参阅

6.1 节介绍异步方法单元测试的基础知识。

6.4 数据流网格的单元测试

问题

程序中有一个数据流网格，需要对它进行正确性验证。

解决方案

数据流网格是独立的：有自己的生命周期，并且本质上就是异步的。自然而然，它的测试方法就是使用异步的单元测试。下面的单元测试验证 4.6 节中的自定义数据流块：

```
[TestMethod]
public async Task MyCustomBlock_AddsOneToDataItems()
{
```

```
    var myCustomBlock = CreateMyCustomBlock();

    myCustomBlock.Post(3);
    myCustomBlock.Post(13);
    myCustomBlock.Complete();

    Assert.AreEqual(4, myCustomBlock.Receive());
    Assert.AreEqual(14, myCustomBlock.Receive());
    await myCustomBlock.Completion;
}
```

可惜的是，对错误进行单元测试就没那么简单了。这是因为在数据流网格中，异常信息在块之间传递时会被一层一层地封装在另一个 AggregateException 中。下面的例子使用了一个辅助方法，以确保一个异常在丢弃数据之后，再在自定义块之间传递。

```
[TestMethod]
public async Task MyCustomBlock_Fault_DiscardsDataAndFaults()
{
    var myCustomBlock = CreateMyCustomBlock();

    myCustomBlock.Post(3);
    myCustomBlock.Post(13);
    myCustomBlock.Fault(new InvalidOperationException());

    try
    {
        await myCustomBlock.Completion;
    }
    catch (AggregateException ex)
    {
        AssertExceptionIs<InvalidOperationException>(
            ex.Flatten().InnerException, false);
    }
}

public static void AssertExceptionIs<TException>(Exception ex,
    bool allowDerivedTypes = true)
{
    if (allowDerivedTypes && !(ex is TException))
        Assert.Fail("Exception is of type " + ex.GetType().Name + ", but "
        + typeof(TException).Name + " or a derived type was expected.");
    if (!allowDerivedTypes && ex.GetType() != typeof(TException))
        Assert.Fail("Exception is of type " + ex.GetType().Name + ", but "
        + typeof(TException).Name + " was expected.");
}
```

讨论

直接对数据流网格做单元测试是可行的，但有些别扭。如果网格是一个大组件的组成部分，只对这个大组件做单元测试（隐式的测试网格），那样会比较简单。但如果开发可重用的自定义块或网格，那就应该像前面那样做单元测试。

参阅

6.1 节介绍异步方法单元测试的基础知识。

6.5 Rx Observable对象的单元测试

问题

程序中用到了 `IObservable<T>`，需要对这部分程序做单元测试。

解决方案

响应式扩展（Reactive Extension）有很多产生序列的操作符（如 `Return`），还有操作符可把响应式序列转换成普通集合或项目（如 `SingleAsync`）。我们可使用 `Return` 等操作符创建 `Observable` 对象依赖项的存根（stub），用 `SingleAsync` 等操作符来测试输出。

看下面的代码，它把一个 HTTP 服务作为依赖项，并且在调用 HTTP 时使用了一个超时：

```
public interface IHttpService
{
    IObservable<string> GetString(string url);
}

public class MyTimeoutClass
{
    private readonly IHttpService _httpService;

    public MyTimeoutClass(IHttpService httpService)
    {
        _httpService = httpService;
    }

    public IObservable<string> GetStringWithTimeout(string url)
    {
        return _httpService.GetString(url)
            .Timeout(TimeSpan.FromSeconds(1));
    }
}
```

我们要测试的代码是 `MyTimeoutClass`，它消耗一个 Observable 对象依赖项，生成一个 Observable 对象作为输出。

`Return` 操作符创建一个只有一个元素的冷序列（cold sequence），可用它来构建简单的存根（stub）。`SingleAsync` 操作符返回一个 `Task<T>` 对象，该对象在下一个事件到达时完成。`SingleAsync` 可用来做简单的单元测试，如下所示：

```
class SuccessHttpServiceStub : IHttpService
{
    public IObservable<string> GetString(string url)
```

```
    {
        return Observable.Return("stub");
    }
}

[TestMethod]
public async Task MyTimeoutClass_SuccessfulGet_ReturnsResult()
{
    var stub = new SuccessHttpServiceStub();
    var my = new MyTimeoutClass(stub);

    var result = await my.GetStringWithTimeout("http://www.example.com/")
        .SingleAsync();

    Assert.AreEqual("stub", result);
}
```

存根代码中另一个重要操作符是 Throw，它返回一个以错误结束的 Observable 对象。这样我们也可对有错误的场景做单元测试。下面的例子使用了 6.2 节中的辅助方法 ThrowsExceptionAsync：

```
private class FailureHttpServiceStub : IHttpService
{
    public IObservable<string> GetString(string url)
    {
        return Observable.Throw<string>(new HttpRequestException());
    }
}

[TestMethod]
public async Task MyTimeoutClass_FailedGet_PropagatesFailure()
{
    var stub = new FailureHttpServiceStub();
    var my = new MyTimeoutClass(stub);

    await ThrowsExceptionAsync<HttpRequestException>(async () =>
    {
        await my.GetStringWithTimeout("http://www.example.com/")
            .SingleAsync();
    });
}
```

讨论

Return 和 Throw 操作符很适合创建 observable 对象的存根，而要在 async 单元测试中测试 observable 对象，比较容易的方法就是使用 SingleAsync。对于简单的 observable 对象，这两个操作符结合起来使用效果很好。但如果 observable 对象与时间有关，它们就不那么管用了。例如要测试 MyTimeoutClass 类的超时能力，单元测试就必须真正地等待那么长时间。一旦增加更多的单元测试，这种方式就不大合适了。6.6 节介绍一种特殊的方法，Reactive Extensions 可以把时间本身排除在外。

参阅

6.1 节介绍对 `async` 方法做单元测试，这与用 `await SingleAsync` 进行单元测试非常相似。

6.6 节介绍对依赖于时间的 observable 序列做单元测试。

6.6 用虚拟时间测试Rx Observable对象

问题

需要写一个不依赖于时间的单元测试，来测试一个依赖于时间的 observable 对象。如 observable 对象中使用了超时、窗口 / 缓冲、限流 / 抽样等方法，那它就是依赖于时间的。我们要对它们做单元测试，但要求运行时间不能太长。

解决方案

我们当然可以让延迟函数在单元测试中运行。但是这样做会产生两个问题：1) 单元测试的运行时间会很长；2) 因为所有的单元测试是同时运行的，这样做会导致竞态条件，无法预测运行时机。

Rx 库在设计时就考虑到了测试问题。实际上，Rx 库本身就已经过大量单元测试。为了解决上面的问题，Rx 引入了调度器（scheduler）这一概念，每个与时间有关的 Rx 操作都在实现时使用了这个抽象的调度器。

要让 observable 对象便于测试，就要允许调用它的程序指定调度器。例如可以使用 6.5 节的 `MyTimeoutClass`，并加上一个调度器：

```
public interface IHttpService
{
    IObservable<string> GetString(string url);
}

public class MyTimeoutClass
{
    private readonly IHttpService _httpService;

    public MyTimeoutClass(IHttpService httpService)
    {
        _httpService = httpService;
    }

    public IObservable<string> GetStringWithTimeout(string url,
        IScheduler scheduler = null)
    {
        return _httpService.GetString(url)
            .Timeout(TimeSpan.FromSeconds(1), scheduler ?? Scheduler.Default);
```

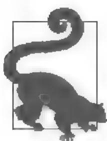
```
        }
    }
```

接下来，修改 HTTP 服务存根，加入调度功能，然后加入一个可变延迟：

```
    private class SuccessHttpServiceStub : IHttpService
    {
        public IScheduler Scheduler { get; set; }
        public TimeSpan Delay { get; set; }

        public IObservable<string> GetString(string url)
        {
            return Observable.Return("stub")
                .Delay(Delay, Scheduler);
        }
    }
```

这样就可以使用 Rx 库中的 TestScheduler 了。可以用 TestScheduler 对（虚拟）时间进行很好的控制。



TestScheduler 在 Rx 中一个单独的 NuGet 包中，需要安装 NuGet 包 Rx-Testing。

用 TestScheduler 可以对时间进行完整的控制，但通常只需要写好代码，然后调用 TestScheduler.Start。在整个测试结束前，Start 方法可以用虚拟的方式推进时间。下面是一个成功测试的简单例子：

```
    [TestMethod]
    public void MyTimeoutClass_SuccessfulGetShortDelay_ReturnsResult()
    {
        var scheduler = new TestScheduler();
        var stub = new SuccessHttpServiceStub
        {
            Scheduler = scheduler,
            Delay = TimeSpan.FromSeconds(0.5),
        };
        var my = new MyTimeoutClass(stub);
        string result = null;

        my.GetStringWithTimeout("http://www.example.com/", scheduler)
            .Subscribe(r => { result = r; });

        scheduler.Start();

        Assert.AreEqual("stub", result);
    }
```

这段代码模拟了 0.5 秒的网络延时。需要强调的是，这个单元测试实际运行时间并没有 0.5 秒。在我的电脑上，做这个测试只需 70 毫秒左右。这个 0.5 秒的延时只不过是虚拟的。另一个值得注意的差别是，这个单元测试不是异步的。因为使用了 `TestScheduler`，所有的测试都会立即完成。

好了，现在使用了调度器，测试超时的情况就很容易了：

```
[TestMethod]
public void MyTimeoutClass_SuccessfulGetLongDelay_ThrowsTimeoutException()
{
    var scheduler = new TestScheduler();
    var stub = new SuccessHttpServiceStub
    {
        Scheduler = scheduler,
        Delay = TimeSpan.FromSeconds(1.5),
    };
    var my = new MyTimeoutClass(stub);
    Exception result = null;

    my.GetStringWithTimeout("http://www.example.com/", scheduler)
        .Subscribe(_ => Assert.Fail("Received value"), ex => { result = ex; });

    scheduler.Start();

    Assert.IsInstanceOfType(result, typeof(TimeoutException));
}
```

再强调一次，运行这个单元测试不需要 1 秒（或 1.5 秒），它会使用虚拟时间立即完成。

讨论

前面我们介绍了 Reactive Extensions 的调度器和虚拟时间的入门知识。Rx 编程和单元测试最好能同时进行。请放心，即使代码越来越复杂，Rx 的测试功能也足以应对了。

`TestScheduler` 还有 `AdvanceTo` 和 `AdvanceBy` 方法，可用来逐步地推进虚拟时间。这些方法在某些场合也许有用，但是应该尽量让每个单元测试只测试一项内容。例如在测试一个超时功能时，可只写一个单元测试，先让 `TestScheduler` 前进一段时间，验证这个超时不会提前发生。然后让 `TestScheduler` 前进预定的超时时间，验证这个超时确实会发生。不过我建议大家尽可能使用独立的单元测试，例如一个单元测试验证超时不会提前发生，另一个单元测试验证超时确实会在后面发生。

参阅

6.5 节介绍对 observable 对象序列做单元测试的基础知识。

第 7 章

互操作

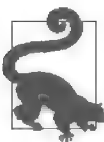
异步、并行、响应式——每种技术都有自己的用武之地，但是结合起来使用会怎样呢？

本章我们来看一下各种互操作的场景，学习如何把这些不同的技术结合起来。我们将了解到这几种技术是互为补充而不是互相排斥的。当它们结合在一起时，在边界上几乎没有什么冲突。

7.1 用async代码封装Async方法与Completed事件

问题

有一种老式的异步编程模式，用的是 OperationAsync 方法和 OperationCompleted 事件。我们希望实现类似的操作，并且用 await 来等待返回的结果。



使用 OperationAsync 和 OperationCompleted 的模式称为基于事件的异步模式（EAP）。我们要把它们封装成返回 Task 对象的方法，并且让它符合基于任务的异步模式（TAP）。

解决方案

可以使用 TaskCompletionSource<TResult> 类创建异步操作的容器。这个类控制一个 Task<TResult> 对象，并且可以在适当的时机完成该任务。

下面的例子定义了一个 WebClient 下载文本的扩展方法。WebClient 类定义了 DownloadStringAsync 和 DownloadStringCompleted，利用这些方法，就可这样定义 DownloadStringTaskAsync 方法：

```
public static Task<string> DownloadStringTaskAsync(this WebClient client,
    Uri address)
{
    var tcs = new TaskCompletionSource<string>();

    // 这个事件处理程序会完成 Task 对象，并自行注销。
    DownloadStringCompletedEventHandler handler = null;
    handler = (_, e) =>
    {
        client.DownloadStringCompleted -= handler;
        if (e.Cancelled)
            tcs.TrySetCanceled();
        else if (e.Error != null)
            tcs.TrySetException(e.Error);
        else
            tcs.TrySetResult(e.Result);
    };

    // 登记事件，然后开始操作。
    client.DownloadStringCompleted += handler;
    client.DownloadStringAsync(address);

    return tcs.Task;
}
```

因为有了 TryCompleteFromEventArgs 扩展方法，若早已经在使用 NuGet 库 Nito.AsyncEx 了，那么实现这种容器就会更简单：

```
public static Task<string> DownloadStringTaskAsync(this WebClient client,
    Uri address)
{
    var tcs = new TaskCompletionSource<string>();

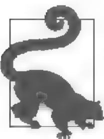
    // 这个事件处理程序会完成 Task 对象，并自行注销。
    DownloadStringCompletedEventHandler handler = null;
    handler = (_, e) =>
    {
        client.DownloadStringCompleted -= handler;
        tcs.TryCompleteFromEventArgs(e, () => e.Result);
    };

    // 登记事件，然后开始操作。
    client.DownloadStringCompleted += handler;
    client.DownloadStringAsync(address);

    return tcs.Task;
}
```

讨论

WebClient 已经定义了 DownloadStringTaskAsync，并且还可以使用更加适合 async 的 HttpClient 类，因此这个实例并没有太大的实用价值。然而，对于那些还没有升级到使用 Task 类的异步代码，可以用这种技术交互。



在新编写代码时都要使用 HttpClient。只有在维护以前遗留的代码时才用 WebClient。

下载文本的 TAP 方法一般应该命名为 OperationAsync（例如 DownloadStringAsync），但本例的情况无法接受这样的命名习惯，因为 EAP 中已经用了这个名称。这时，习惯上把 TAP 方法命名为 OperationTaskAsync（例如 DwonloadStringTaskAsync）。

在封装 EAP 方法时，"启动"方法有可能抛出异常。前面的例子中可能抛出异常的方法就是 DownloadStringAsync。这时开发者需要做出选择：是让异常继续传递，还是捕获异常并调用 TrySetException。在那个位置抛出异常通常是因为使用不当，因此选择哪一种方式都可以，区别不大。

参阅

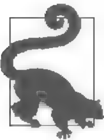
7.2 节对 APM 方法（BeginOperation 和 EndOperation）进行 TAP 封装。

7.3 节对各种类型的通知进行 TAP 封装。

7.2 用async代码封装Begin/End方法

问题

有一种老式的异步编程模式，它使用一对名为 BeginOperation 和 EndOperation 的方法来和表示这个异步操作的 IAsyncResult 接口。我们希望能用 await 来调用这种模式的操作。



使用 BeginOperation 和 EndOperation 的模式称为异步编程模型（APM）。我们要把它们封装成返回 Task 对象的方法，符合基于任务的异步模式（TAP）。

解决方案

封装 APM 最好的办法是使用 TaskFactory 类型的一个 FromAsync 方法。FromAsync 在内部

使用 TaskCompletionSource<Result>，但在封装 APM 时，FromAsync 用起来更方便。

下面的例子定义了一个 WebRequest 的扩展方法，发送一个 HTTP 请求并获取响应。Web Request 类定义了 BeginGetResponse 和 EndGetResponse。我们可以这样定义 GetResponse Async 方法：

```
public static Task<WebResponse> GetResponseAsync(this WebRequest client)
{
    return Task<WebResponse>.Factory.FromAsync(client.BeginGetResponse,
        client.EndGetResponse, null);
}
```

讨论

FromAsync 的重载个数多得一塌糊涂！

通常来讲，最好用例子中的方式调用 FromAsync。首先传入 BeginOperation 方法（不调用）和 EndOperation8 方法（不调用）。接着传入 BeginOperation 所需的全部参数（后面的 AsyncCallback 和 object 参数除外）。最后传入 null。

这里要特别指出的是，不要在调用 FromAsync 之前调用 BeginOperation。调用 FromAsync，并让用 BeginOperation 方法返回的 IAsyncOperation 作为参数，这样也是可以的，但是 FromAsync 会采用效率较低的实现方式。

也许你会感到奇怪，怎么推荐的这个模式总是在最后传入 null。FromAsync 是在 .NET 4.0 版本中和 Task 类一起被引入的，当时还没有关键字 async。当时在异步回调函数中普遍使用 state 对象，Task 类通过 AsyncState 成员来支持这种调用方式。新的 async 模式就再也不需要 state 对象了。

参阅

7.3 节介绍为任何类型的通知编写 TAP 封装器。

7.3 用async代码封装所有异步操作

问题

有一个不常见或不标准的异步操作或事件，我们希望能用 await 来调用。

解决方案

任何情况下都可以用 TaskCompletionSource<T> 类来构造 Task<T> 对象。使用 Task CompletionSource<T> 时，Task 对象的完成可以有三种不同的方式：成功得到结果、出错、被取消。

在 async 出现前，微软推荐另外两种异步编程模式：APM（见 7.2 节）和 EAP（见 7.1 节）。但 APM 和 EAP 都相当繁琐，也经常难以得到正确结果。因此产生了一种非官方的通行做法，即使用回调函数，就像下面的方法：

```
public interface IMyAsyncHttpService
{
    void DownloadString(Uri address, Action<string, Exception> callback);
}
```

此类方法遵循这样的通行流程：DwonloadString 启动（异步地）下载，下载完成时，包含返回信息或异常信息的 callback 被触发。通常 callback 是在后台线程中被触发的。

这个非标准类型的异步编程方法，也能用 TaskCompletionSource<T> 进行封装，能让 await 进行正常调用：

```
public static Task<string> DownloadStringAsync(
    this IMyAsyncHttpService httpService, Uri address)
{
    var tcs = new TaskCompletionSource<string>();
    httpService.DownloadString(address, (result, exception) =>
    {
        if (exception != null)
            tcs.TrySetException(exception);
        else
            tcs.TrySetResult(result);
    });
    return tcs.Task;
}
```

讨论

这种模式结合 TaskCompletionSource<T>，可以封装任何异步方法，不管它有多么不标准。首先创建 TaskCompletionSource<T> 实例。接着准备一个回调函数，以便 TaskCompletion Source<T> 能顺利完成它的 Task 对象。然后开始真正的异步操作。最后返回附属于 TaskCompletionSource<T> 的 Task<T>。

关于这种模式有一点十分重要，就是必须确保 TaskCompletionSource<T> 总是处于完成状态。尤其要仔细地检查一下错误处理过程，并且确保 TaskCompletionSource<T> 会正常完成。在最后一个例子中，异常被显式传递进回调函数，因此程序中不需要有 catch 块。但是一些非标准的模式会要求在回调函数中捕获异常，并把异常信息放在 TaskCompletion Source<T> 中。

参阅

7.1 节介绍对 EAP 模式的成员（OperationAsync、OperationCompleted）进行 TAP 封装。

7.2 节介绍对 APM 模式的成员（BeginOperation，EndOperation）进行 TAP 封装。

7.4 用async代码封装并行代码

问题

希望用 `await` 调用计算密集型的处理过程。采用这种做法后，在等待并行处理完成时能避免 UI 线程阻塞。

解决方案

`Parallel` 类和并行 `LINQ` 利用线程池做并行处理。它们也会把调用线程作为并行处理的线程之一，因此从 UI 线程调用并行方法时，UI 会在并行处理结束前一直保持停止响应状态 。

要让 UI 保持响应的话，就可将并行处理过程封装进 `Tast.Run`，并使用 `await`：

```
await Task.Run(() => Parallel.ForEach(...));
```

这个方法的关键，是并行代码把调用线程也看做是用于并行处理的线程池的一部分。并行 `LINQ` 和 `Parallel` 类都是这样来处理的。

讨论

这个方法很简单，却经常被忽视。通过使用 `Task.Run`，所有的并行处理过程都推给了线程池。`Task.Run` 返回一个代表并行任务的 `Task` 对象，UI 线程可以（异步地）等待它完成。

这个方法只能用于 UI 代码。在服务器端（例如 ASP.NET）很少用并行处理。如果一定要在服务器端使用并行处理过程，那也应该直接调用它，而不能把它推给线程池。

参阅

第 3 章介绍了并行代码的基础知识。

第 2 章介绍了异步代码的基础知识。

7.5 用async代码封装Rx Observable对象

问题

希望用 `await` 来处理一个可观察流。

解决方案

首先需要确定事件流中的哪一个事件是需要关注的。通常有几种情况：

- 事件流结束前的最后一个事件；
- 下一个事件；
- 所有事件。

要捕获事件流的最后一个事件，可用 await 调用 LastAsync 方法的结果，或者直接对 Observable 对象进行 await：

```
IObservable<int> observable = ...;
int lastElement = await observable.LastAsync();
// 或者 int lastElement = await observable;
```

在 await 调用 Observable 对象或 LastAsync 时，代码（异步地）等待事件流完成，然后返回最后一个元素。在内部，await 实际是在订阅事件流。

使用 FirstAsync 可捕获事件流中的下一个事件。本例中 await 订阅事件流，然后在第一个事件到达后立即结束（并退订）：

```
IObservable<int> observable = ...;
int nextElement = await observable.FirstAsync();
```

使用 ToList 可捕获事件流中的所有事件：

```
IObservable<int> observable = ...;
IList<int> allElements = await observable.ToList();
```

讨论

Rx 库提供了 await 处理事件流所需的全部工具。唯一的难点是我们必须考虑这些方法是否会一直等待，直到事件流结束。本节的例子中，LastAsync、ToList 和直接使用 await 会等待事件流结束，FirstAsync 只会等待下一个事件到达。

如果这些例子不能满足需求，还可以考虑使用完整的 LINQ 功能和新版 Rx 控制器。如果只要异步地等待某些元素而不是整个事件流完成，可以使用 Task 和 Buffer 等操作符。

某些和 await 一起使用的操作符（如 FirstAsync 和 LastAsync）并不会真正地返回 Task<T> 对象。如果要使用 Task.WhenAll 或 Task.WhenAny，就需要有实际的 Task<T> 对象。可在 Observable 对象上调用 ToTask，以得到这个 Task<T> 对象，该对象代表着事件流结束时的最后一个值。

参阅

7.6 节介绍在可观察流中使用异步代码。

7.7 节介绍用可观察流作为数据流块的输入（该数据流块可以异步运行）。

5.3 节对可观察流进行窗口和缓冲操作。

7.6 用Rx Observable对象封装async代码

问题

需要把一个异步操作与一个 observable 对象结合。

解决方案

任何异步操作都可看作一个满足以下条件之一的可观察流：

- 生成一个元素后就完成；
- 发生错误，不生成任何元素。

Rx 库中有一个将 `Task<T>` 转换成 `IObservable<T>` 的简单方法。下面的代码启动一个异步的网页下载过程，并把它作为一个 observable 序列：

```
var client = new HttpClient();
IObservable<HttpResponseMessage> response =
    client.GetAsync("http://www.example.com/")
    .ToObservable();
```

使用 `ToObservable` 前必须调用 `async` 方法并转换成 `Task` 对象。

另一个办法是调用 `StartAsync`。`StartAsync` 也会立即调用 `async` 方法，但它支持取消功能：如果订阅已被处理，这个 `async` 方法就会被取消：

```
var client = new HttpClient();
IObservable<HttpResponseMessage> response = Observable
    .StartAsync(token => client.GetAsync("http://www.example.com/", token));
```

`ToObservable` 和 `StartAsync` 都会立即启动异步操作，而不会等待订阅。如果要让 observable 对象在接受订阅后才启动操作，可使用 `FromAsync`（跟 `StartAsync` 一样，它也支持取消功能）：

```
var client = new HttpClient();
IObservable<HttpResponseMessage> response = Observable
    .FromAsync(token => client.GetAsync("http://www.example.com/", token));
```

`FromAsync` 和 `ToObservable`、`StartAsync` 有着显著的区别。`ToObservable` 和 `StartAsync` 都返回一个 observable 对象，表示一个已经启动的异步操作。`FromAsync` 在每次被订阅时都会启动一个全新独立的异步操作。

最后，如果要在源事件流中每到达一个事件就启动一个异步操作，就可使用 `SelectMany` 的

特殊重载。SelectMany 也支持取消功能。

下面的例子使用一个已有的 URL 事件流，在每个 URL 到达时发出一个请求：

```
IObservable<string> urls = ...
var client = new HttpClient();
IObservable<HttpResponseMessage> responses = urls
    .SelectMany((url, token) => client.GetAsync(url, token));
```

讨论

响应式扩展在 async 引进之前就存在了，但后来增加了上述（和其他）操作符，以便与 async 代码互通。即使能够用其他 Rx 操作符实现同样的功能，我还是建议大家使用上面提到的操作符。

参阅

7.5 节介绍在异步代码中使用可观察流。

7.7 节介绍用数据流块（可包含异步代码）作为可观察流的来源。

7.7 Rx Observable对象和数据流网格

问题

同一个项目中，一部分使用了 Rx Observable 对象，一部分使用了数据流网格，现在需要它们能互相沟通。

Rx Observable 对象和数据流网格有各自的用途，也存在一些概念上的重叠。本节说明它们能互相配合得很好，因此可以在项目中的不同部分选用最合适的工具。

解决方案

首先，我们考虑把数据流块用作可观察流的输入。下面的代码创建一个缓冲块（它不处理数据），然后调用 AsObservable 来创建一个缓冲块到 Observable 对象的接口：

```
var buffer = new BufferBlock<int>();
IObservable<int> integers = buffer.AsObservable();
integers.Subscribe(data => Trace.WriteLine(data),
    ex => Trace.WriteLine(ex),
    () => Trace.WriteLine("Done"));

buffer.Post(13);
```

缓冲数据流块和可观察流都会正常完成或者出错，AsObservable 方法会把数据流块的完成信息（或出错信息）转化为可观察流的完成信息。如果数据流块出错并抛出异常，这个异常信息在传递给可观察流时，会被封装在 AggregateException 对象中。这种方式与互相连接的数据流块之间传递错误的方式有些相似。

如果使用一个网格并把它作为可观察流的目的，情况只会稍微复杂一点。下面的代码调用 AsObserver 让一个块订阅一个可观察流：

```
IObservable<DateTimeOffset> ticks =
    Observable.Interval(TimeSpan.FromSeconds(1))
    .Timestamp()
    .Select(x => x.Timestamp)
    .Take(5);

var display = new ActionBlock<DateTimeOffset>(x => Trace.WriteLine(x));
ticks.Subscribe(display.AsObserver());

try
{
    display.Completion.Wait();
    Trace.WriteLine("Done.");
}
catch (Exception ex)
{
    Trace.WriteLine(ex);
}
```

跟前面一样，可观察流的完成信息会转化为块的完成信息，可观察流的错误信息会转化为块的错误信息。

讨论

数据流块和可观察流的很多基础概念是一样的。它们都能传递数据，都能处理完成信息和错误信息。它们是为不同的场景设计的。TPL 数据流针对异步和并行混合编程，而 Rx 针对响应式编程。但概念上的重叠部分具有足够的兼容性，两者能配合得很好、很自然。

参阅

7.5 节介绍在异步代码中使用可观察流。

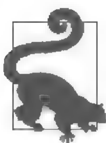
7.6 节介绍在可观察流中使用异步代码。

第 8 章

集合

使用合适的集合对于并发程序来说是必不可少的。这里我们不讨论标准的集合，例如 List<T>，因为这些大家早就很熟悉了。本章介绍一些专门用于并发或异步开发的新集合。

不可变集合是永远不会改变的集合。这种集合看起来好像没什么用处，但实际上它们用途很广泛，甚至能用在单线程、非并发的程序中。只读操作（如枚举）直接访问不可变集合实例。写入操作（如增加一个项目）会返回一个新的不可变集合实例，而不是修改原来的实例。乍一听上去，这种做法浪费存储空间，但不可变集合之间通常共享了大部分存储空间，因此其实浪费并不大。并且不可变集合有个优势，多个线程访问是安全的。因为是无法修改的，所以是线程安全的。



不可变集合在 NuGet 包 Microsoft.Bcl.Immutable 中。

在编写本书时，不可变集合还是一个新事物。但所有新开发中都应该考虑使用不可变集合，除非确实需要可变的集合。如果你对不可变集合还不熟悉，哪怕你并不需要栈或队列，也建议你从 8.1 节开始阅读，因为那一节介绍了所有不可变集合都遵循的一些通用模式。

如果要用很多已有的元素来构建不可变集合，可使用一些高效的特殊方法来完成。在这几节的例子中，每次只会添加一个元素。MSDN 文档中有关于如何快速构建不可变集合的描述。表 8-1 是各平台对不可变集合的支持情况。

表8-1：各平台对不可变集合的支持

平　　台	ImmutableStack<T>等
.NET4.5	√
.NET4.0	×
Mono iOS/Droid	√
Windows Store	√
Windows Phone Apps 8.1	√
Windows Phone SL 8.0	√
Windows Phone SL 7.1	×
Silverlight 5	×

线程安全集合是可同时被多个线程修改的可变集合。线程安全集合混合使用了细粒度锁定和无锁技术，以确保线程被阻塞的时间最短（通常情况下是根本不阻塞）。对很多线程安全集合进行枚举操作时，内部创建了该集合的一个快照（snapshot），并对这个快照进行枚举操作。线程安全集合的主要优点是多个线程可以安全地对其进行访问，而代码只会被阻塞很短的时间，或根本不阻塞。表 8-2 是各平台对线程安全集合的支持情况。

表8-2：各平台对线程安全集合的支持

平　　台	ConcurrentDictionary<Tkey TValue>等
.NET 4.5	√
.NET 4.0	√
Mono iOS/Droid	√
Windows Store	√
Windows Phone Apps 8.1	√
Windows Phone SL 8.0	×
Windows Phone SL 7.1	×
Silverlight 5	×

生产者 / 消费者集合是一种可变集合，这类集合的设计带有特殊的目的：支持（可能有多个）生产者向集合推送项目，同时支持（可能有多个）消费者从集合取走项目。它们在生产者代码和消费者代码之间架设了桥梁，并且可通过设置来限制集合中的项目数量。生产者 / 消费者集合可以有阻塞或异步的 API。例如，集合为空时，一个阻塞的生产者 / 消费者集合会阻塞正在调用的消费者线程，直到有一个项目加入集合它才停止。但是一个异步的生产者 / 消费者集合会使消费者线程进行异步等待，直到加入另一个项目。表 8-3 是各平台对生产者 / 消费者集合的支持情况。

表8-3：各平台对生产者/消费者集合的支持

平　　台	BlockingCollection<T>	BufferBlock<T>	AsyncProducerConsumerQueue<T>	AsyncCollection<T>
.NET 4.5	√	√	√	√
.NET 4.0	√	×	√	√

（续）

平　　台	BlockingCollection<T>	BufferBlock<T>	AsyncProducerConsumerQueue<T>	AsyncCollection<T>
Mono iOS/Droid	√	√	√	√
Windows Store	√	√	√	√
Windows Phone Apps 8.1	√	√	√	√
Windows Phone SL 8.0	×	√	√	×
Windows Phone SL 7.1	×	×	√	×
Silverlight 5	×	×	√	×



AsyncProducerConsumerQueue<T> 和 AsyncCollection<T> 在 NuGet 包 Nito.AsyncEx 中。BufferBlock<T> 在 NuGet 包 Microsoft.Tpl.Dataflow 中。

本章用到了多种不同的生产者 / 消费者集合，它们各有不同的优势。表 8-4 可用于队生产者 / 消费者集合的选择。

表8-4：生产者/消费者集合

功　　能	BlockingCollection<T>	BufferBlock<T>	AsyncProducerConsumerQueue<T>	AsyncCollecton<T>
队列语法	√	√	√	√
栈 / 包语法	√	×	×	√
同步 API	√	√	√	√
异步 API	×	√	√	√
支持移动平台	部分	部分	√	部分
通过微软测试	√	√	×	×

8.1　不可变栈和队列

问题

需要一个不会经常修改、可以被多个线程安全访问的栈或队列。

例如，可用来表示一系列操作的队列，可用来表示一系列取消操作的栈。

解决方案

不可变栈和队列是最简单的不可变集合。它们的特征与标准的 Stack<T> 和 Queue<T> 非常

相似。在性能上，不可变栈和队列与标准栈和队列具有一样的时间复杂度。但是在需要频繁修改的简单情况下，标准栈和队列的速度更快。

栈是“后进先出”的数据结构。下面的代码创建一个空的不可变栈，接着压入两个项目，枚举这些项目，最后弹出项目：

```
var stack = ImmutableStack<int>.Empty;
stack = stack.Push(13);
stack = stack.Push(7);

// 先显示“7”，接着显示“13”。
foreach (var item in stack)
    Trace.WriteLine(item);

int lastItem;
stack = stack.Pop(out lastItem);
// lastItem == 7
```

在上面的例子中，值得注意的是，程序对局部变量 stack 进行了覆盖。不可变集合采用的模式是返回一个修改过的集合，原始的集合引用是不变化的。这意味着，如果引用了特定的不可变集合的实例，它是不会变化的，具体可看下面的例子：

```
var stack = ImmutableStack<int>.Empty;
stack = stack.Push(13);
var biggerStack = stack.Push(7);

// 先显示“7”，接着显示“13”。
foreach (var item in biggerStack)
    Trace.WriteLine(item);

// 只显示“13”。
foreach (var item in stack)
    Trace.WriteLine(item);
```

两个栈实际上在内部共享了存储项目 13 的内存。这种实现方式的效率很高，并且可以很方便地创建当前状态的快照。每个不可变集合的实例都是绝对线程安全的，但也能在单线程程序中使用。根据我的经验，如果代码功能增加，或者需要存储很多快照并希望它们能尽可能多地共享内存，那不可变集合就特别有用。

队列与栈类似，但队列是“先进先出”的数据结构。下面的代码创建一个空的不可变队列，然后加入两个项目，枚举这些项目，最后把项目从队列中取出：

```
var queue = ImmutableQueue<int>.Empty;
queue = queue.Enqueue(13);
queue = queue.Enqueue(7);

// 先显示“13”，接着显示“7”。
foreach (var item in queue)
    Trace.WriteLine(item);
```

```
int nextItem;
queue = queue.Dequeue(out nextItem);
// 显示“13”。
Trace.WriteLine(nextItem);
```

讨论

本节介绍了两个最简单的不可变集合：栈和队列。也介绍了几个适用于所有不可变集合的重要设计理念。

- 不可变集合的一个实例是永远不改变的。
- 因为不会改变，所以是绝对线程安全的。
- 对不可变集合使用修改方法时，返回修改后的集合。

不可变集合非常适用于共享状态，但不适合用来做交换数据的通道。特别是在线程间的通信中不要使用不可变队列，可以改用生产者 / 消费者队列，以获得更好的效果。



`ImmutableStack<T>` 和 `ImmutableQueue<T>` 在 NuGet 包 `Microsoft.Bcl.Immutable` 中。

参阅

8.6 节介绍线程安全（阻塞）的可变队列。

8.7 节介绍线程安全（阻塞）的可变栈。

8.8 节介绍兼容异步操作的可变队列。

8.9 节介绍兼容异步操作的可变栈。

8.10 节介绍阻塞 / 异步的可变队列。

8.2　不可变列表

问题

需要一个这样的数据结构：支持索引，不经常修改，可以被多个线程安全访问。

列表是多功能数据结构，可用于所有类型的程序状态。

解决方案

不可变列表确实可以索引，但需要注意性能问题。不能简单地用它来替代 List<T>。

ImmutableList<T> 支持与 List<T> 类似的方法，看下面的例子：

```
var list = ImmutableList<int>.Empty;
list = list.Insert(0, 13);
list = list.Insert(0, 7);

// 先显示“13”，接着显示“7”。
foreach (var item in list)
    Trace.WriteLine(item);

list = list.RemoveAt(1);
```

不可变列表的内部是用二叉树组织数据的。这么做是为了让不可变列表的实例之间共享的内存最大化。这导致 ImmutableList<T> 和 List<T> 在常用操作上有性能上的差别（参见表 8-5）。

表8-5：不可变列表的性能差异

操　作	List<T>	ImmutableList<T>
Add	平摊 $O(1)$	$O(\log N)$
Insert	$O(N)$	$O(\log N)$
RemoveAt	$O(N)$	$O(\log N)$
Item[index]	$O(1)$	$O(\log N)$

需要特别注意的是，ImmutableList<T> 索引操作的时间复杂度是 $O(\log N)$，大家可能会误以为是 $O(1)$。如果在已有的代码中用 ImmutableList<T> 来代替 List<T>，需要弄清楚算法逻辑是如何访问集合中元素的。

这意味着应该尽量使用 foreach 而不是用 for。对 ImmutableList<T> 进行 foreach 循环的耗时是 $O(N)$，而对同一个集合进行 for 循环的耗时是 $O(N*\log N)$：

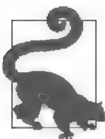
```
// 遍历 ImmutableList<T> 的最好方法。
foreach (var item in list)
    Trace.WriteLine(item);

// 这个方法运行正常，但速度会慢得多。
for (int i = 0; i != list.Count; ++i)
    Trace.WriteLine(list[i]);
```

讨论

ImmutableList<T> 是一种优秀的多功能数据结构，但因为有性能上的差异，不能盲目地用它来代替 List<T>。默认情况下一般使用 List<T>，就是说，通常都要使用 List<T>，除非确实需要使用其他集合。ImmutableList<T> 的使用就没那么普遍，需要仔细考虑其他不可

变集合，并选择一个最合适的。



ImmutableList<T> 在 NuGet 包 Microsoft.Bcl.Immutable 中。

参阅

8.1 节介绍不可变栈和队列，它们与列表类似，都只允许访问指定的元素。

MSDN 中有关于 ImmutableList<T>.Builder 的文档，这是一种快速构建不可变列表的方法。

8.3 不可变Set集合

问题

需要一个这样的数据结构：不需要存放重复内容，不经常修改，可以被多个线程安全访问。

例如，文件的词汇索引就是使用 Set 集合的一个实例。

解决方案

有两种不可变 Set 集合类型：ImmutableHashSet<T> 只是一个不含重复元素的集合，ImmutableSortedSet<T> 是一个已排序的不含重复元素的集合。这两种 Set 集合类型都有相似的接口：

```
var hashSet = ImmutableHashSet<int>.Empty;
hashSet = hashSet.Add(13);
hashSet = hashSet.Add(7);

// 显示“7”和“13”，次序不确定。
foreach (var item in hashSet)
    Trace.WriteLine(item);

hashSet = hashSet.Remove(7);
```

只是已排序的 Set 集合可以使用索引访问，类似于列表：

```
var sortedSet = ImmutableSortedSet<int>.Empty;
sortedSet = sortedSet.Add(13);
sortedSet = sortedSet.Add(7);

// 先显示“7”，接着显示“13”。
foreach (var item in hashSet)
```

```
        Trace.WriteLine(item);
    var smallestItem = sortedSet[0];
    // smallestItem == 7

    sortedSet = sortedSet.Remove(7);
```

未排序的 Set 集合和已排序的 Set 集合，两者的性能差不多（见表 8-6）。

表8-6：不可变Set集合的性能

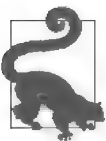
操　　作	ImmutableHashSet<T>	ImmutableSortedSet<T>
Add	$O(\log N)$	$O(\log N)$
Remove	$O(\log N)$	$O(\log N)$
Item[index]	不可用	$O(\log N)$

如果不是一定要排序，我建议大家使用未排序的 Set 集合。有些类型只支持判断是否相等，而不支持比较大小，因此未排序 Set 集合支持的类型比已排序 Set 集合要多得多。

关于已排序 Set 集合有一点要特别注意，它索引操作的时间复杂度是 $O(\log N)$，而不是 $O(1)$，这跟 8.2 节中 `ImmutableList<T>` 的情况类似。这意味着它们适用同样的警告：使用 `ImmutableSortedSet<T>` 时，应该尽量用 `foreach` 而不是用 `for`。

讨论

不可变 Set 集合是非常实用的数据结构，但是填充较大不可变 Set 集合的速度会很慢。大多数不可变集合有特殊的构建方法，可以先快速地以可变方式构建，然后转换成不可变集合。这种构建方法可用于很多不可变集合，但我发现对不可变 Set 集合是最有用的。



`ImmutableHashSet<T>` 和 `ImmutableSortedSet<T>` 在 NuGet 包 `Microsoft.Bcl.Immutable` 中。

参阅

8.7 节介绍线程安全的可变包，与 Set 集合类似。

8.9 节介绍兼容异步操作的可变包。

MSDN 中有关于 `ImmutableHashSet<T>.Builder` 的文档，这是一种快速构建不可变 Set 集合的方法。

MSDN 中有关于 `ImmutableSortedSet<T>.Builder` 的文档，这是一种快速构建已排序不可变 Set 集合的方法。

8.4 不可变字典

问题

需要一个不经常修改且可被多个线程安全访问的键/值集合。

例如需要存储查询集合中的参考数据。这些参考数据很少修改但需要被不同的线程访问。

解决方案

有两种不可变字典类型：ImmutableDictionary<TKey,TValue> 和 ImmutableSortedDictionary<TKey,TValue>。也许你已经猜到了，ImmutableSortedDictionary 确保它的元素是已经排序的，而 ImmutableDictionary 的元素次序是无法预知的。

这两种集合类型的成员非常相似：

```
var dictionary = ImmutableDictionary<int, string>.Empty;
dictionary = dictionary.Add(10, "Ten");
dictionary = dictionary.Add(21, "Twenty-One");
dictionary = dictionary.SetItem(10, "Diez");

// 显示“10Diez”和“21Twenty-One”，次序不确定。
foreach (var item in dictionary)
    Trace.WriteLine(item.Key + item.Value);

var ten = dictionary[10];
// ten == "Diez"

dictionary = dictionary.Remove(21);
```

注意 SetItem 的用法。在可变字典中，可以使用这样的语句：dictionary[key] = item。但是不可变字典必须返回一个更新后的不可变字典，因此用 SetItem 方法来代替：

```
var sortedDictionary = ImmutableSortedDictionary<int, string>.Empty;
sortedDictionary = sortedDictionary.Add(10, "Ten");
sortedDictionary = sortedDictionary.Add(21, "Twenty-One");
sortedDictionary = sortedDictionary.SetItem(10, "Diez");

// 先显示“10Diez”，接着显示“21Twenty-One”。
foreach (var item in sortedDictionary)
    Trace.WriteLine(item.Key + item.Value);

var ten = sortedDictionary[10];
// ten == "Diez"

sortedDictionary = sortedDictionary.Remove(21);
```

未排序字典和已排序字典在性能上差别不大，但是我建议大家使用未排序字典，除非是必

须排序（见表 8-7）。未排序字典的速度稍微快一点。而且未排序字典可以使用任何键类型，而已排序字典要求键的类型必须是完全可比较的。

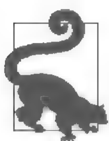
表8-7：不可变字典的性能

操　作	ImmutableDictionary<TK,TV>	ImmutableSortedDictionary<TK,TV>
Add	$O(\log N)$	$O(\log N)$
SetItem	$O(\log N)$	$O(\log N)$
Item[key]	$O(\log N)$	$O(\log N)$
Remove	$O(\log N)$	$O(\log N)$

讨论

根据经验，字典是处理应用状态时很普遍又实用的工具。它能用在任何类型的键 / 值或查询。

跟其他不可变集合一样，不可变字典有一个在元素较多时进行快速构建的机制。例如，想要在启动时装载初始参考数据，就可以使用这种构建机制构造出初始的不可变字典。相反，如果参考数据是在程序运行时逐步构建的，那可以使用常规的 Add 方法。



ImmutableDictionary<TK, TV> 和 ImmutableSortedDicationary<TK, TV> 在 NuGet 包 Microsoft.Bcl.Immutable 中。

参阅

8.5 节介绍线程安全的可变字典。

MSDN 中关于 ImmutableDictionary<TK,TV>.Builder 的文档，介绍了快速填充不可变字典的方法。

MSDN 中关于 ImmutableSortedDictionary<TK,TV>.Builder 的文档，介绍了快速填充已排序的不可变字典的方法。

8.5　线程安全字典

问题

需要有一个键 / 值集合，多个线程同时读写时仍能保持同步。

例如一个简单的驻留内存缓存。

解决方案

.NET 中的 ConcurrentDictionary<TKey, TValue> 类是数据结构中的精品。它是线程安全的，混合使用了细粒度锁定和无锁技术，以确保绝大多数情况下能进行快速访问。

熟悉它的 API 确实需要一定的时间。因为要处理多线程的并发访问，它与标准的 Dictionary<TKey,TValue> 类完全不同。但是一旦学完本节内容，你就会发现 ConcurrentDictionary<TKey, TValue> 是最实用的集合类型之一。

首先我们来看如何在集合中写入一个数据。要设置一个键对应的值，可使用 AddOrUpdate，如：

```
var dictionary = new ConcurrentDictionary<int, string>();
var newValue = dictionary.AddOrUpdate(0,
    key => "Zero",
    (key, oldValue) => "Zero");
```

AddOrUpdate 方法有些复杂，这是因为这个方法必须执行多个步骤，具体步骤取决于并发字典的当前内容。方法的第一个参数是键。第二个参数是一个委托，它把键（本例中为 0）转换成添加到字典的值（本例中为“Zero”）。只有当字典中没有这个键时，这个委托才会运行。第三个参数也是一个委托，它把键（0）和原来的值转换成字典中修改后的值（“Zero”）。只有当字典中已经存在这个键时，这个委托才会运行。AddOrUpdate 返回这个键对应的新值（与其中一个委托返回的值相同）。

有一点会让大家非常吃惊：为使并发字典正常运行，AddOrUpdate 可能要多次调用其中一个（或两个）委托。这种情况很少，但确实会发生。因此这些委托必须简单、快速，并且不能有副作用。也就是说，这些委托只能创建新的值，不能修改程序中其他变量。这个原则适用于所有 ConcurrentDictionary<TKey,TValue> 的方法所使用的委托。

因为要处理线程安全方面的所有问题，这部分内容是比较难的。其余的 API 就简单多了。

其实还有几种方法可以向字典中添加数据。一个简便方法是使用索引语法：

```
// 使用与上一个例子同样的“字典”。
// 添加（或修改）键 0，对应值“Zero”。
dictionary[0] = "Zero";
```

索引语法的功能相对较弱，它不支持根据当前的值进行修改的方法。如果把数据直接存入字典，那使用这种语法会更简单，效果也不错。

来看一下如何读取值。很简单，使用 TryGetValue 就行：

```
// 使用与前面一样的“字典”。
string currentValue;
bool keyExists = dictionary.TryGetValue(0, out currentValue);
```

如果字典中存在这个键，TryGetValue 返回 true，并填写输出的变量。如果键不存在，TryGetValue 返回 false。也可以使用索引语法来读取值，但我发现这不怎么实用，因为如果在键不存在，就会抛出异常。需要注意，有多个线程在对并发字典进行读取、修改、添加和删除值的操作。不试着读取一下，很多情况下是很难确定某个键是否存在的。

删除值的操作跟读取一样简单：

```
// 使用与前面一样的"字典"。
string removedValue;
bool keyExisted = dictionary.TryRemove(0, out removedValue);
```

TryRemove 几乎和 TryGetValue 一样，除了（当然了）它是进行删除操作的，如果键存在，就删除"键 / 值"对。

讨论

我觉得 ConcurrentDictrionary<TKey,TValue> 是一个很好的类，主要是因为有功能特别强大的 AddOrUpdate 方法。但是它并不适合于所有场合。如果多个线程读写一个共享集合，使用 ConcurrentDictrionary<TKey,TValue> 是最合适的。如果不会频繁修改（很少修改），那更适合使用 ImmutableDictionary<TKey, TValue>。

ConcurrentDictrionary<TKey,TValue> 最适合用在需要共享数据的场合，即多个线程共享同一个集合。如果一些线程只添加元素，另一些线程只移除元素，那最好使用生产者 / 消费者集合。

ConcurrentDictrionary<TKey,TValue> 并不是唯一的线程安全集合。BCL 库还提供了 ConcurrentStack<T>、ConcurrentQueue<T> 和 ConcurrentBag<T>。不过它们很少单独使用，一般只是用来实现生产者 / 消费者集合，本章后面会介绍。

参阅

8.4 节介绍了不可变字典。如字典内容极少修改，不可变字典则是最理想的选择。

8.6　阻塞队列

问题

需要有一个管道，在线程之间传递消息或数据。例如，一个线程正在装载数据，装载的同时把数据压进管道。与此同时，另一个线程在管道的接收端接收并处理数据。

解决方案

.NET 的 BlockingCollection<T> 类可用作这种管道。BlockingCollection<T> 默认是阻塞队列，具有“先进先出”的特征。

因为阻塞队列要被多个线程共用，通常把它定义成私有和只读：

```
private readonly BlockingCollection<int> _blockingQueue =
    new BlockingCollection<int>();
```

通常一个线程要么向集合中添加项目，要么移除项目，但不会两者都做。添加项目的线程为生产者线程，移除项目的线程为消费者线程。

生产者线程通过调用 Add 方法来添加项目，在添加完成（即所有项目都已经添加完毕）后调用 CompleteAdding 方法。这个方法通知集合，表示“没有更多的项目需要添加了”，然后该集合会通知消费者线程。

在下面的简单例子中，生产者添加两个项目，然后做“完成”的标志：

```
_blockingQueue.Add(7);
_blockingQueue.Add(13);
_blockingQueue.CompleteAdding();
```

消费者线程通常运行一个循环，等待下一个项目然后处理该项目。若使用上述生产者代码并放在独立的线程里（例如用 Task.Run），就可以用下面的方法使用这些项目了：

```
// 先显示“7”，后显示“13”。
foreach (var item in _blockingQueue.GetConsumingEnumerable())
    Trace.WriteLine(item);
```

如果想要有多个消费者，可以在多个线程中同时调用 GetConsumingEnumerable。每个项目只会传给其中的一个线程。当集合处理完毕后，这个枚举过程也结束。

除非能保证消费者的速度总是比生产者快，使用这种管道时，都要考虑一旦生产者比消费者快，会发生什么情况。如果生产项目的速度比消费快，就需要对队列进行限流。BlockingCollection<T> 类可以很方便地实现限流功能，可以在创建队列时设置限流的项目个数。下面的例子把集合的项目数量限制为 1 个：

```
BlockingCollection<int> _blockingQueue = new BlockingCollection<int>(
    boundedCapacity: 1);
```

这样，同样的生产者代码的运行方式就会不同，具体看代码中的注释：

```
// 这个添加过程立即完成。
_blockingQueue.Add(7);

// 7 被移除后，添加 13 才会完成 。
```

```
_blockingQueue.Add(13);

_blockingQueue.CompleteAdding();
```

讨论

前面例子中的消费者线程都使用了 GetConsumingEnumerable 方法。这是最常用的做法，但也可使用 Take 方法，它每次只会消费一个项目，而不是用一个循环使用所有的项目。

如果有独立的线程（如线程池线程）作为生产者或消费者，阻塞队列就是一个十分不错的选择。如果要以异步方式访问管道，例如 UI 线程作为消费者，用阻塞队列就不大合适了。8.8 节会介绍异步队列。

如果准备在程序中使用这样的管道，可考虑改用 TPL 数据流库。在很多情况下，使用 TPL 数据流会比自己创建管道和后台线程要简单。特别是 BufferBlock<T> 可以作为阻塞队列使用。不过，并不是每个平台都支持 TPL 数据流的，对那些不支持 TPL 数据流的平台来说，选择阻塞队列是很适用的。

如果要有最好的跨平台支持，也可以使用 AsyncEx 库中的 AsyncProducerConsumerQueue<T>，它可以用作阻塞队列。表 8-8 列出了各平台对阻塞队列的支持情况。

表8-8：各平台对阻塞队列的支持情况

平　台	BlockingCollection<T>	BufferBlock<T>	AsyncProducerConsumerQueue<T>
.NET 4.5	√	√	√
.NET 4.0	√	×	√
Mono iOS/Droid	√	√	√
Windows Store	√	√	√
Windows Phone Apps 8.1	√	√	√
Windows Phone SL 8.0	×	√	√
Windows Phone SL 7.1	×	×	√
Silverlight 5	×	×	√

参阅

8.7 节介绍了阻塞栈和包，它们也是类似的管道，但不是“先进先出”的。

8.8 节介绍具有异步 API 而不是阻塞 API 的队列。

8.10 节介绍既有异步 API 又有阻塞 API 的队列。

8.7 阻塞栈和包

问题

需要有一个管道，在线程之间传递消息或数据，但不想（或不需要）这个管道使用“先进先出”的语义。

解决方案

在默认情况下，.NET 中的 BlockingCollection<T> 用作阻塞队列，但它也可以作为任何类型的生产者 / 消费者集合。BlockingCollection<T> 实际上是对线程安全集合进行了封装，实现了 IProducerConsumerCollection<T> 接口。

因此可以在创建 BlockingCollection<T> 实例时指明规则，可选择后进先出（栈）或无序（包），如下例所示：

```
BlockingCollection<int> _blockingStack = new BlockingCollection<int>(
    new ConcurrentStack<int>());

BlockingCollection<int> _blockingBag = new BlockingCollection<int>(
    new ConcurrentBag<int>());
```

有一点很重要并需要引起注意，就是这时已经出现了有关项目次序的竞态条件。如果先运行生产者代码，后运行消费者代码，那项目的次序就和使用栈完全一样：

```
// 生产者代码
_blockingStack.Add(7);
_blockingStack.Add(13);
_blockingStack.CompleteAdding();

// 消费者代码
// 先显示“13”，后显示“7”。
foreach (var item in _blockingStack.GetConsumingEnumerable())
    Trace.WriteLine(item);
```

但是如果生产者代码和消费者代码在不同的线程中（这是常见情况），消费者会一直取得最近加入的项目。例如，生产者加入 7，接着消费者取走 7，生产者加入 13，接着消费者取走 13。消费者在返回第一个项目前，不会等待生产者调用 CompleteAdding。

讨论

阻塞队列有关限流的注意事项，同样适用于阻塞栈和包。如果生产者的速度比消费者快，又要限制阻塞栈和包对内存的使用，就可以使用 8.6 节讨论过的限流方法。

本节的消费者代码使用 GetConsumingEnumerable，这是最常用的做法。但也可使用 Take 方

法，它每次只会消费一个项目，而不是用一个循环消费掉所有的项目。

如果不用阻塞方式，而是要用异步方式访问共享的栈或包（例如，UI 线程作为消费者），请看 8.9 节。

参阅

8.6 节介绍了阻塞队列，它的使用比阻塞栈或包要广泛得多。

8.9 节介绍了异步栈和包。

8.8　异步队列

问题

需要有一个管道，在代码的各个部分之间以后进先出的方式传递消息或数据。

例如，一段代码在加载数据，并向管道推送数据。同时 UI 线程在接收并显示数据。

解决方案

只需要一个带有异步 API 的队列。在 .NET 核心框架中没有这样的类，但是在 NuGet 中有好几个可以选择。

第一个选择是使用 TPL 数据流库的 `BufferBlock<T>`。下面的例子展示了声明 `BufferBlock<T>` 实例的方法、生产者代码和消费者代码的样式：

```
BufferBlock<int> _asyncQueue = new BufferBlock<int>();

// 生产者代码
await _asyncQueue.SendAsync(7);
await _asyncQueue.SendAsync(13);
_asyncQueue.Complete();

// 消费者代码
// 先显示“7”，后显示“13”。
while (await _asyncQueue.OutputAvailableAsync())
    Trace.WriteLine(await _asyncQueue.ReceiveAsync());
```

`BufferBlock<T>` 本身也支持限流，详见 8.10 节。

例子中消费者代码使用了 `OutputAvailableAsync`，这个方法其实只能用在仅有一个消费者的情况下。如果有多个消费者，即使队列中只有一个项目，`OutputAvailableAsync` 也可能对每个消费者都返回 `true`。如果队列中项目都取完了，`DequeueAsync` 会抛出 `InvalidOperationException` 异常。因此在有多个消费者时，消费者的代码通常更像下面这样：

```
while (true)
{
    int item;
    try
    {
        item = await _asyncQueue.ReceiveAsync();
    }
    catch (InvalidOperationException)
    {
        break;
    }
    Trace.WriteLine(item);
}
```

如果所用的平台支持 TPL 数据流，我建议大家使用 BufferBlock<T> 的方案。可惜并不是所有平台都支持 TPL 数据流。如果平台不支持 BufferBlock<T>，那可以用 NuGet 包 Nito.AsyncEx 中的 AsyncProducerConsumerQueue<T> 类。它的 API 与 BufferBlock<T> 类似，但不完全相同：

```
AsyncProducerConsumerQueue<int> _asyncQueue
    = new AsyncProducerConsumerQueue<int>();

// 生产者代码
await _asyncQueue.EnqueueAsync(7);
await _asyncQueue.EnqueueAsync(13);
await _asyncQueue.CompleteAdding();

// 消费者代码
// 先显示“7”，后显示“13”。
while (await _asyncQueue.OutputAvailableAsync())
    Trace.WriteLine(await _asyncQueue.DequeueAsync());
```

AsyncProducerConsumerQueue<T> 具有限流功能，如果生产者的运行速度可能比消费者快，这个功能就是必需的。只要在构造队列时使用适当的参数即可：

```
AsyncProducerConsumerQueue<int> _asyncQueue
    = new AsyncProducerConsumerQueue<int>(maxCount: 1);
```

这样，同样的生产者代码会以异步方式正确地等待了：

```
// 这个添加过程会立即完成。
await _asyncQueue.EnqueueAsync(7);

// 这个添加过程会（异步地）等待，直到 7 被移除，
// 然后才会加入 13。
await _asyncQueue.EnqueueAsync(13);

await _asyncQueue.CompleteAdding();
```

例子中的消费者代码也使用 OutputAvailableAsync，并且也有跟 BufferBlock<T> 同样的问题。AsyncProducerConsumerQueue<T> 类提供了 TryDequeueAsync 成员，可以用来避免冗长的消费者代码。如果有多个消费者，消费者代码通常像这样：

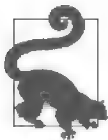
```
while (true)
{
    var dequeueResult = await _asyncQueue.TryDequeueAsync();
    if (!dequeueResult.Success)
        break;
    Trace.WriteLine(dequeueResult.Item);
}
```

讨论

BufferBlock<T> 和 AsyncProducerConsumerQueue<T> 相比，我更推荐使用前者，仅仅是因为对 BufferBlock<T> 进行的测试要完整得多。但是有很多平台不支持 BufferBlock<T>，尤其是一些较老的平台（见表 8-9）。

表8-9：各平台对异步队列的支持情况

平　台	BufferBlock<T>	AsyncProducerConsumerQueue<T>
.NET 4.5	√	√
.NET 4.0	×	√
Mono iOS/Droid	√	√
Windows Store	√	√
Windows Phone Apps 8.1	√	√
Windows Phone SL 8.0	√	√
Windows Phone SL 7.1	×	√
Silverlight 5	×	√



BufferBlock<T> 类 在 NuGet 包 Microsoft.Tpl.Dataflow 中。AsyncProducerConsumerQueue<T> 类在 NuGet 包 Nito.AsyncEx 中。

参阅

8.6 节介绍了阻塞语义（而不是异步语义）的生产者 / 消费者队列。

8.10 节介绍了同时具有阻塞语义和异步语义的生产者 / 消费者队列。

8.7 节介绍了异步栈和包，可用于需要类似的管道，但不要先进先出语义的场合。

8.9　异步栈和包

问题

需要一个管道，用来在程序的各个部分之间传递消息或数据，但不希望（或不需要）这个

管道使用“先进先出”的语义。

解决方案

Nito.AsyncEx 库提供了 AsyncCollection<T> 类，它默认表现为异步队列，但也可以作为任何类型的生产者 / 消费者集合。AsyncCollection<T> 对 IProducerConsumerCollection<T> 进行了封装。AsyncCollection<T> 相当于是异步版的 .NET BlockingCollection<T> 类。

AsyncCollection<T> 支持后进先出（栈）或无序（包）的语义，取决于构造函数中传入什么集合：

```
AsyncCollection<int> _asyncStack = new AsyncCollection<int>(
    new ConcurrentStack<int>());
AsyncCollection<int> _asyncBag = new AsyncCollection<int>(
    new ConcurrentBag<int>());
```

注意在栈的项目次序上有竞态条件。如果所有生产者完成后，消费者才开始运行，那项目的次序就像一个普通的栈：

```
// 生产者代码
await _asyncStack.AddAsync(7);
await _asyncStack.AddAsync(13);
await _asyncStack.CompleteAddingAsync();

// 消费者代码
// 先显示“13”，后显示“7”。
while (await _asyncStack.OutputAvailableAsync())
    Trace.WriteLine(await _asyncStack.TakeAsync());
```

但是，当生产者和消费者都并发运行时（这是常见情况），消费者总是会得到最近加入的项目。这导致这个集合从整体上看不像是一个栈。当然了，包是根本没有次序的。

AsyncCollection<T> 具有限流功能，如果生产者添加项目到集合的速度可能比消费者从集合取走项目的速度快，这个功能就是必需的。只要在构造集合时使用合适的值就行了：

```
AsyncCollection<int> _asyncStack = new AsyncCollection<int>(
    new ConcurrentStack<int>(), maxCount: 1);
```

这样，同样的生产者代码会根据需要异步地等待了：

```
// 这个添加过程会立即完成。
await _asyncStack.AddAsync(7);

// 这个添加（异步地）等待，直到 7 被移除，
// 然后才会加入 13。
await _asyncStack.AddAsync(13);

await _asyncStack.CompleteAddingAsync();
```

例子中的消费者代码使用了 OutputAvailableAsync，这个方法同样有如 8.8 节描写的那种限制。如果有多个消费者，消费者代码通常更像这样：

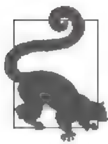
```
while (true)
{
    var takeResult = await _asyncStack.TryTakeAsync();
    if (!takeResult.Success)
        break;
    Trace.WriteLine(takeResult.Item);
}
```

讨论

AsyncCollection<T> 实际上只是异步版的 BlockingCollection<T> 类，并且支持的平台也一样（见表 8-10）。

表8-10：各平台对栈和包的支持情况

平　台	BlockingCollection<T>（阻塞）	AsyncCollection<T>（异步）
.NET 4.5	✓	✓
.NET 4.0	✓	✓
Mono iOS/Droid	✓	✓
Windows Store	✓	✓
Windows Phone Apps 8.1	✓	✓
Windows Phone SL 8.0	×	×
Windows Phone SL 7.1	×	×
Silverlight 5	×	×



AsyncCollection<T> 类在 NuGet 包 Nito.AsyncEx 中。

参阅

8.8 节介绍了异步队列，它的使用比异步栈或包要广泛得多。

8.7 节介绍了异步（阻塞）栈和包。

8.10　阻塞/异步队列

问题

需要一个管道，用“先进先出”的方式在程序的各部分之间传递消息或数据。并且要有足

够的灵活性，能以同步或异步方式来处理生产者终端或消费者终端。

例如，一个后台线程在装载数据并把数据压入管道，我们希望当管道太满时该线程能同步地阻塞。同时，UI 线程在从管道接收数据，我们希望这个线程异步地从管道拉取数据，以便 UI 保持响应。

解决方案

我们已经看了 8.6 节中的阻塞队列、8.8 节中的异步队列，但是还有几种同时支持阻塞 API 和异步 API 的队列。

首先是 NuGet 库 TPL 数据流中的 BufferBlock<T> 和 ActionBlock<T>。BufferBlock<T> 能很方便地用作异步的生产者 / 消费者队列（详见 8.8 节）：

```
BufferBlock<int> queue = new BufferBlock<int>();

// 生产者代码
await queue.SendAsync(7);
await queue.SendAsync(13);
queue.Complete();

// 单个消费者时的代码
while (await queue.OutputAvailableAsync())
    Trace.WriteLine(await queue.ReceiveAsync());

// 多个消费者时的代码
while (true)
{
    int item;
    try
    {
        item = await queue.ReceiveAsync();
    }
    catch (InvalidOperationException)
    {
        break;
    }

    Trace.WriteLine(item);
}
```

BufferBlock<T> 也有用于生产者和消费者的同步 API：

```
BufferBlock<int> queue = new BufferBlock<int>();

// 生产者代码
queue.Post(7);
queue.Post(13);
queue.Complete();
```

```
// 消费者代码
while (true)
{
    int item;
    try
    {
        item = queue.Receive();
    }
    catch (InvalidOperationException)
    {
        break;
    }

    Trace.WriteLine(item);
}
```

但是使用了 BufferBlock<T> 的消费者代码是十分笨拙的，因为这不是“数据流的风格”。TPL 数据流库包含几个能互相连接的块，可以定义一个响应式网格。在本例中，可以用 ActionBlock<T> 定义一个带有特定动作的生产者 / 消费者队列：

```
// 消费者代码被传给队列的构造函数
ActionBlock<int> queue = new ActionBlock<int>(item => Trace.WriteLine(item));

// 异步的生产者代码
await queue.SendAsync(7);
await queue.SendAsync(13);

// 同步的生产者代码
queue.Post(7);
queue.Post(13);
queue.Complete();
```

如果你选定的平台不支持 TPL 数据流库，那可以使用 Nito.AsyncEx 中的 AsyncProducerConsumerQueue<T> 类，它同时也支持同步和异步方法：

```
AsyncProducerConsumerQueue<int> queue = new AsyncProducerConsumerQueue<int>();

// 异步的生产者代码
await queue.EnqueueAsync(7);
await queue.EnqueueAsync(13);

// 同步的生产者代码
queue.Enqueue(7);
queue.Enqueue(13);

queue.CompleteAdding();

// 单个消费者时的异步代码
while (await queue.OutputAvailableAsync())
    Trace.WriteLine(await queue.DequeueAsync());

// 多个消费者时的异步代码
```

```
while (true)
{
    var result = await queue.TryDequeueAsync();
    if (!result.Success)
        break;
    Trace.WriteLine(result.Item);
}

// 同步的消费者代码
foreach (var item in queue.GetConsumingEnumerable())
    Trace.WriteLine(item);
```

讨论

虽然 AsyncProducerConsumerQueue<T> 支持更多的平台，我仍建议大家尽可能使用 BufferBlock<T> 或 ActionBlock<T>，因为对 TPL 数据流库做过的测试比 Nito.AsyncEx 更加完整。

像 AsyncProducerConsumerQueue<T> 这样的 TPL 数据流块也支持限流功能，可以通过传递构造函数的参数来实现此功能。如果生产者压入项目的速度比消费者消耗项目的速度块，就会导致程序占用大量的内存，这种情况下必须使用限流功能。表 8-11 列出了个平台对同步 / 异步队列的支持情况。

表8-11：各平台对同步/异步队列的支持情况

平台	BufferBlock<T>和ActionBlock<T>	AsyncProducerConsumerQueue<T>
.NET 4.5	√	√
.NET 4.0	×	√
Mono iOS/Droid	√	√
Windows Store	√	√
Windows Phone Apps 8.1	√	√
Windows Phone SL 8.0	√	√
Windows Phone SL 7.1	×	√
Silverlight 5	×	√



BufferBlock<T> 类和 ActionBlock<T> 类在 NuGet 包 Microsoft.Tpl.Dataflow 中。AsyncProducerConsumerQueue<T> 类在 NuGet 包 Nito.AsyncEx 中。

参阅

8.6 节介绍了阻塞的生产者 / 消费者队列。

8.8 节介绍了异步的生产者 / 消费者队列。

4.4 节介绍了数据流块的限流功能。

第 9 章 取消

.NET 4.0 框架引入了详尽的、精心设计的取消功能。取消采用协作方式，即可以请求某段代码取消，但不能强制它取消。如果某段代码本身不支持取消，就无法请求它取消运行。基于这个原因，建议大家在编写代码时尽量支持取消。

取消是一种信号，包含两个不同的方面：触发取消的源头和响应取消的接收器。在 .NET 中源头是 `CancellationTokenSource`，接收器是 `CancellationToken`。本章将介绍这两方面常规用法，还将介绍如何与非标准的取消模式进行互操作。

取消被看作是一种特殊类型的错误。通常被取消的代码会抛出类型为 `OperationCanceledException`（或者它的子类，如 `TaskCanceledException`）的异常。调用的代码用这种方式确认取消信号已被接收。

为了表明某个方法支持取消，需要用一个 `CancellationToken` 作为该方法的参数。这个参数通常放在最后，除非该方法也支持进度报告（见 2.3 节）。也可考虑提供一个重载函数或一个参数默认值，供不需要取消的程序使用：

```
public void CancelableMethodWithOverload(CancellationToken cancellationToken)
{
    // 这里放代码
}

public void CancelableMethodWithOverload()
{
    CancelableMethodWithOverload(CancellationToken.None);
}
```

```
public void CancelableMethodWithDefault(
    CancellationToken cancellationToken = default(CancellationToken))
{
    // 这里放代码
}
```

CancellationToken.None 是一个等同于 default(CancellationToken) 的特殊值，表示这个方法是永远不会被取消的。如果启动这个操作后不准备取消，就可以在调用时使用这个值。

9.1 发出取消请求

问题

有一段可取消的代码（使用了 CancellationToken），需要把它取消。

解决方案

CancellationToken 来源于 CancellationTokenSource 类。CancellationToken 只是让代码能够响应取消请求，用 CancellationTokenSource 的用户可发出取消请求。

多个 CancellationTokenSource 互相之间是独立的（除非把它们连接起来，具体操作将在 9.8 节介绍）。CancellationTokenSource 的 Token 属性返回它的 CancellationToken，Cancel 方法发出真正的取消请求。

下面的代码展示了如何创建一个 CancellationTokenSource 实例和如何使用 Token 和 Cancel。用简短的例子更容易说明问题，因此代码使用了 async 方法。同样的 Token/Cancel 组合可以用来取消所有类型的代码：

```
void IssueCancelRequest()
{
    var cts = new CancellationTokenSource();
    var task = CancelableMethodAsync(cts.Token);

    // 到这里，操作已经启动。

    // 发出取消请求。
    cts.Cancel();
}
```

在前面的例子中，当任务启动后就把 Task 变量忽略了。在实际项目的开发中，这个变量可能会被存储起来并使用 await 等待，以便最终用户能看到运行结果。

在取消任务运行时通常会产生竞态条件。如果在发出取消请求的时刻，被取消的代码即将完成，若来不及检查取消标记，那它就会正常地结束。在取消代码时实际上会有三种可能性：响应取消请求（抛出 OperationCanceledException），正常结束，或者出现跟取消无关

的错误并结束（抛出其他异常）。

下面的例子与前面类似，但使用了 await，说明了三种可能的结果：

```
async Task IssueCancelRequestAsync()
{
    var cts = new CancellationTokenSource();
    var task = CancelableMethodAsync(cts.Token);

    // 这里，操作在正常运行。

    // 发出取消请求。
    cts.Cancel();

    //（异步地）等待操作结束。
    try
    {
        await task;
        // 如运行到这里，说明在取消请求生效前，操作正常完成 。
    }
    catch (OperationCanceledException)
    {
        // 如运行到这里，说明操作在完成前被取消。
    }
    catch (Exception)
    {
        // 如运行到这里，说明在取消请求生效前，操作出错并结束。
        throw;
    }
}
```

创建 CancellationTokenSource 实例和发出取消请求，这两步一般放在不同方法中。CancellationTokenSource 实例一旦销毁就无法恢复。如果需要另一个取消标记源，就必须创建另一个实例。下面是一个更接近实际开发的 GUI 界面的例子，用一个按钮启动异步操作，另一个按钮用来取消这个操作。程序会禁用或启用“开始”和“取消”这两个按钮，以保证同一时间内只有一个操作：

```
private CancellationTokenSource _cts;

private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    StartButton.IsEnabled = false;
    CancelButton.IsEnabled = true;

    try
    {
        _cts = new CancellationTokenSource();
        var token = _cts.Token;
        await Task.Delay(TimeSpan.FromSeconds(5), token);
        MessageBox.Show("Delay completed successfully.");
    }
    catch (OperationCanceledException)
```

```
        {
            MessageBox.Show("Delay was canceled.");
        }
        catch (Exception)
        {
            MessageBox.Show("Delay completed with error.");
            throw;
        }
        finally
        {
            StartButton.IsEnabled = true;
            CancelButton.IsEnabled = false;
        }
    }

    private void CancelButton_Click(object sender, RoutedEventArgs e)
    {
        _cts.Cancel();
    }
```

讨论

本节用一个 GUI 程序作为最接近实际开发的例子，但不要误以为只有用户界面程序才能用取消。取消也可用在服务器程序中，例如 ASP.NET 中有一个表示请求超时的取消标记。服务器端的取消标记源确实很少，但是没有理由说不能用。我就曾在 ASP.NET 将要卸载应用程序域时，用一个 CancellationTokenSource 来请求取消。

参阅

9.4 节介绍如何向 async 代码传递取消标记。

9.5 节介绍如何向并发代码传递取消标记。

9.6 节介绍如何向响应式代码传递取消标记。

9.7 节介绍如何向数据流网格传递取消标记。

9.2 通过轮询响应取消请求

问题

在代码的循环中实现取消。

解决方案

代码中正在运行一个循环时，就没有更低级别的 API 来接收 CancellationToken 了。这时

可以周期性地检查标记是否已被取消。下面的代码在运行计算密集型的循环时，周期性地监测标记：

```
public int CancelableMethod(CancellationToken cancellationToken)
{
    for (int i = 0; i != 100; ++i)
    {
        Thread.Sleep(1000); // 这里做一些计算工作。
        cancellationToken.ThrowIfCancellationRequested();
    }
    return 42;
}
```

如果循环很密集（即循环体的运行速度很快），就需要限制检查取消标记的频率。在确定最佳做法之前，通常要对此类修改前后的程序性能进行评估。下面的代码与上一个例子类似，但是循环速度更快、次数更多，因此对检查标记的频率进行了限制：

```
public int CancelableMethod(CancellationToken cancellationToken)
{
    for (int i = 0; i != 100000; ++i)
    {
        Thread.Sleep(1); // 这里做一些计算工作。
        if (i % 1000 == 0)
            cancellationToken.ThrowIfCancellationRequested();
    }
    return 42;
}
```

限制多少才是合适的，这完全取决于代码工作量的大小，以及对响应速度的要求。

讨论

大多数情况下，只需要把 CancellationToken 传递给下一层就行了。9.4 节至 9.7 节有这方面的例子。只有要求在循环代码中支持取消时，才需要轮询检查标记。

CancellationToke 类还有一个成员 IsCancellationRequested，它会在标记被取消后返回 true。有些人使用这个成员来响应取消请求，即通常返回一个默认值或 null。不过我一般并不推荐这种做法。处理取消请求的标准模式是抛出一个 OperationCanceledException 异常，这个过程由 ThrowIfCancellationRequested 来负责。如果更进一步的代码要捕获这个异常，并且处理结果为 null 的情况，那这种做法就没问题。但如果代码完全控制了 CancellationToken，就该遵循处理取消请求的标准模式。如果确实不想遵循标准模式，至少要有详细的描述。

通过轮询取消标记使 ThrowIfCancellationRequested 起效，代码中必须以固定间隔调用这个方法。还有一种做法是注册一个回调函数，收到取消请求时会被调用。这种用回调函数的做法更像是与其他取消体系的互操作，因此我们把它放到 9.9 节。

参阅

9.4 节介绍如何向 async 代码传递取消标记。

9.5 节介绍如何向并发代码传递取消标记。

9.6 节介绍如何向响应式代码传递取消标记。

9.7 节介绍如何向数据流网格传递取消标记。

9.9 节介绍如何用回调函数代替轮询，来响应取消请求。

9.1 节介绍如何发出取消请求。

9.3 超时后取消

问题

需要让一些代码在发生超时后停止运行。

解决方案

发生超时后，取消便是一种很自然的解决方案。超时只是一种取消请求的类型。需要取消的代码仅仅监视取消标记，不管取消的类型，代码不知道也不关心取消的来源是一个定时器。

.NET 4.5 为取消标记源添加了几个便捷的方法，这些方法会基于定时器自动发出取消请求。可以把超时数据传给构造函数：

```
async Task IssueTimeoutAsync()
{
    var cts = new CancellationTokenSource(TimeSpan.FromSeconds(5));
    var token = cts.Token;
    await Task.Delay(TimeSpan.FromSeconds(10), token);
}
```

还有一个方案，如果已经有了一个 CancellationTokenSource 实例，可以对该实例启动一个超时：

```
async Task IssueTimeoutAsync()
{
    var cts = new CancellationTokenSource();
    var token = cts.Token;
    cts.CancelAfter(TimeSpan.FromSeconds(5));
    await Task.Delay(TimeSpan.FromSeconds(10), token);
}
```

.NET 4.0 不支持采用构造函数的方案，但是该平台的 NuGet 包 `Microsoft.Bcl.Async` 支持 `CancelAfter` 方法。

讨论

只要执行代码时用到了超时，就该使用 `CancellationTokenSource` 和 `CancelAfter`（或者用构造函数）。虽然还有其他途径可实现这个功能，但是使用现有的取消体系是最简单也是最高效的。

别忘了被取消的代码需要监视取消标记。不支持取消的代码，是不可能被轻易取消的。

参阅

9.4 节介绍如何向 `async` 代码传递取消标记。

9.5 节介绍如何向并发代码传递取消标记。

9.6 节介绍如何在响应式代码中使用取消标记。

9.7 节介绍如何向数据流网格传递取消标记。

9.4 取消async代码

问题

需要让 `async` 代码支持取消。

解决方案

要使异步代码支持取消，最简单的办法就是把 `CancellationToken` 传递给下一层代码。这个例子执行一个异步的延时，然后返回一个值。它只是把标记传递给 `Task.Delay`，就实现了对取消的支持：

```
public async Task<int> CancelableMethodAsync(CancellationToken cancellationToken)
{
    await Task.Delay(TimeSpan.FromSeconds(2), cancellationToken);
    return 42;
}
```

很多异步 API 都支持 `CancellationToken`，因此在程序中实现取消是很容易的，只需要将标记传递下去即可。这有一个通用的准则，如果一个方法调用了支持 `CancellationTaken` 的 API，那这个方法也要支持 `CancellationToken` 并把它传给每个支持它的 API。

讨论

很可惜，有一些方法是不支持取消的。没有简单的解决方案可处理这种情况。要安全地停止这种“专横的”代码是不可能的，除非把它封装在一个独立的可执行程序中。碰到此类情况，你可以选择忽略它的返回结果，假装已经取消了这个操作。

只要有可能，每个方法都要支持取消以供调用者选择使用。这是因为只有较低层次的代码正确地支持取消，较高层次的代码才有可能正确地支持取消。因此在编写自己的 async 方法时，要尽最大可能支持取消。你无法预料哪个较高层次的方法会调用你的方法，而该方法可能需要支持取消。

参阅

9.1 节介绍如何发出取消请求。

9.3 节介绍把取消用作超时。

9.5 取消并行代码

问题

需要让并行代码支持取消。

解决方案

要支持取消，最简单的方法是把 CancellationToken 传递进并行代码。并行方法使用 Parallel Options 实例来支持取消。可以用下面的方法设置 ParallelOptions 实例的 CancellationToken：

```
static void RotateMatrices(IEnumerable<Matrix> matrices, float degrees,
    CancellationToken token)
{
    Parallel.ForEach(matrices,
        new ParallelOptions { CancellationToken = token },
        matrix => matrix.Rotate(degrees));
}
```

另一个方法在循环体中直接监视 CancellationToken：

```
static void RotateMatrices2(IEnumerable<Matrix> matrices, float degrees,
    CancellationToken token)
{
    // 警告：不推荐使用，原因见后面。
    Parallel.ForEach(matrices, matrix =>
```

```
        {
            matrix.Rotate(degrees);
            token.ThrowIfCancellationRequested();
        });
    }
```

但是第二种方法更麻烦，用起来也不协调。使用第二种方法时，并行循环会把 OperationCanceledException 封装进 AggregateException。另外，如果把 CancellationToken 传入到 ParallelOptions 的实例中，Parallel 类会做出更加智能化的判断，确定检查标记的频率。因此，把标记作为参数传入是最好的做法。

并行 LINQ（PLINQ）本身也支持取消，可用 WithCancellation 操作符：

```
static IEnumerable<int> MultiplyBy2(IEnumerable<int> values,
    CancellationToken cancellationToken)
{
    return values.AsParallel()
        .WithCancellation(cancellationToken)
        .Select(item => item * 2);
}
```

讨论

在并行处理中支持取消，对于提高用户体验非常重要。程序在做并行处理时，至少会在短时间内使用大量的 CPU。当 CPU 使用率很高时，即使没有妨碍同一电脑上的其他程序，用户也会注意到。因此建议大家在做并行计算（或其他 CPU 密集型程序）时一定要支持取消，即使维持 CPU 高使用率的总时间不是很长。

参阅

9.1 节介绍如何发出取消请求。

9.6 取消响应式代码

问题

需要让响应式代码支持取消。

解决方案

响应式扩展库有一个订阅可观察事件流的概念。要停止对事件流的订阅，只需在代码中释放订阅接口。一般情况下要在逻辑上停止对事件流的订阅，用这种方法就足够了。例如下面的代码，按一个按钮后就订阅鼠标点击事件流，按另一个按钮后就停止订阅：

```
private IDisposable _mouseMovesSubscription;

private void StartButton_Click(object sender, RoutedEventArgs e)
{
    var mouseMoves = Observable
        .FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Select(x => x.EventArgs.GetPosition(this));
    _mouseMovesSubscription = mouseMoves.Subscribe(val =>
    {
        MousePositionLabel.Content = "(" + val.X + ", " + val.Y + ")";
    });
}

private void CancelButton_Click(object sender, RoutedEventArgs e)
{
    if (_mouseMovesSubscription != null)
        _mouseMovesSubscription.Dispose();
}
```

但是，在 Rx 框架中融合被广泛使用的 CancellationTokenSource/CancellationToken 体系其实是很方便的。本节的其余部分介绍 Rx 与 CancellationToken 交互的方法。

首先来看一个主要场景：异步代码封装了可观察流代码。7.5 节详细介绍了这种情况，现在我们要增加它对 CancellationToken 的支持。一般来说最简单的做法是：用响应式操作符实现所有操作，然后调用 ToTask 把最后一个作为结果的元素转换成支持 await 的 Task 对象。下面的代码展示用异步方式使用队列中的最后一个元素：

```
CancellationToken cancellationToken = ...
IObservable<int> observable = ...
int lastElement = await observable.TakeLast(1).ToTask(cancellationToken);
// 或者 int lastElement = await observable.ToTask(cancellationToken);
```

使用第一个元素的方法也类似，只需要在调用 ToTask 前改一下 observable：

```
CancellationToken cancellationToken = ...
IObservable<int> observable = ...
int firstElement = await observable.Take(1).ToTask(cancellationToken);
```

用异步方式把整个 observable 序列转换成 task 对象，也很类似：

```
CancellationToken cancellationToken = ...
IObservable<int> observable = ...
IList<int> allElements = await observable.ToList().ToTask(cancellationToken);
```

最后我们来看相反的情况。我们刚看了 Rx 代码响应 CancellationToken 的几种方法，也就是说，CancellationTokenSource 的取消请求被转化为一个停止订阅指令（释放订阅接口）。我们也可以走另一个途径：把发出取消请求作为对释放订阅接口的响应。

正如 7.6 节讲的，操作符 FromAsync、StartAsync、SelectMany 都支持取消。在绝大部分情况下这已经够用了。Rx 也支持 CancellationDisposable 类，可以这样直接使用：

```
using (var cancellation = new CancellationDisposable())
{
    CancellationToken token = cancellation.Token;
    // 把这个标记传给会对它作出响应的方法。
}
// 到这里，这个标记已经是取消的。
```

讨论

Rx 有它自己关于取消的理念，那就是：停止订阅。.NET 4.0 引入了通用的取消框架。本节介绍了几种让 Rx 与这种通用框架有机融合的方法。如果某段代码中只用到了 Rx，那就使用 Rx 的"订阅 / 停止订阅"体系。只有在边界上才引入 CancellationToken，以保持代码清晰。

参阅

7.5 节介绍了把 Rx 代码封装成异步代码（不支持取消）。

7.6 节介绍了把异步代码封装成 Rx 代码（支持取消）。

9.1 节介绍了如何发送取消请求。

9.7 取消数据流网格

问题

需要让数据流网格支持取消。

解决方案

在自己的代码中支持取消，最好的方法就是把 CancellationToken 传递给支持取消的 API。数据流网格的每一个块都支持取消，可在 DataflowBlockOptions 中设置。要让自定义数据流块也支持取消，只需要在块的参数中设置 CancellationToken 属性：

```
IPropagatorBlock<int, int> CreateMyCustomBlock(
    CancellationToken cancellationToken)
{
    var blockOptions = new ExecutionDataflowBlockOptions
    {
        CancellationToken = cancellationToken
    };
```

```
        var multiplyBlock = new TransformBlock<int, int>(item => item * 2,
            blockOptions);
        var addBlock = new TransformBlock<int, int>(item => item + 2,
            blockOptions);
        var divideBlock = new TransformBlock<int, int>(item => item / 2,
            blockOptions);

        var flowCompletion = new DataflowLinkOptions
        {
            PropagateCompletion = true
        };
        multiplyBlock.LinkTo(addBlock, flowCompletion);
        addBlock.LinkTo(divideBlock, flowCompletion);

        return DataflowBlock.Encapsulate(multiplyBlock, divideBlock);
    }
```

这个例子中网格的每一个块都使用了 CancellationToken。这并不是十分必要的。因为完成信息也在块之间传递，可以只在第一块使用 CancellationToken，然后让它在块之间传递。取消被认为是一种特殊形式的错误信息，而错误信息会在管道中传递下去，因此管道中的其他块也会产生错误并结束。但是当我们取消一个网格时，会希望每一个块同时被取消。因此通常是在每一个块中设置 CancellationToken。

讨论

在数据流网格中，取消过程不会有任何缓冲。块被取消后就会清除所有输入数据并停止接收新项目。因此如果取消一个运行中的块，数据就会丢失。

参阅

9.1 节介绍如何发送取消请求。

9.8 注入取消请求

问题

某一个层次的代码需要响应取消请求，同时它本身也要向下一层代码发出取消请求。

解决方案

.NET 4.0 的取消体系本身就有这种功能，即连接的取消标记。在创建一个取消标记源时，可把它连接到一个（或多个）已有的标记。建立连接后，这些已有标记中的任何一个被取消，新建的标记源中的标记也会被取消。也可以显式地取消这个标记源。

下面的代码执行一个异步的 HTTP 请求。传入该方法的标记代表用户发出的取消请求，而这个方法也对 HTTP 请求使用了一个超时：

```
async Task<HttpResponseMessage> GetWithTimeoutAsync(string url,
    CancellationToken cancellationToken)
{
    var client = new HttpClient();

    using (var cts = CancellationTokenSource
        .CreateLinkedTokenSource(cancellationToken))
    {
        cts.CancelAfter(TimeSpan.FromSeconds(2));
        var combinedToken = cts.Token;
        return await client.GetAsync(url, combinedToken);
    }
}
```

如果用户取消了原有的 `cancellationToken`，或者关联的标记源被 `CancelAfter` 取消，这个创建的 `combinedToken` 就会被取消。

讨论

例子中只用了一个 `CancellationToken` 对象，但是 `CreateLinkedTokenSource` 方法的参数可以是任意多个取消标记。我们可以用它来创建一个组合标记，实现具有某种逻辑的取消功能。例如，ASP.NET 有一个表示请求超时的标记（`HttpRequest.TimedOutToken`）和一个表示用户断开连接的标记（`HttpResponse.ClientDisconnectedToken`）。可以在代码中创建一个连接的标记，对这些取消请求中的任意一个做出响应。

需要注意已连接的取消标记源的生命周期。前面的例子代表了常见的情况，一个或多个取消标记传入方法，然后被连接在一起并作为组合标记继续传下去。注意例子中的代码使用了 `using` 语句，当操作完成时（不需要继续使用组合标记了）会释放已连接的取消标记源。假定不释放已连接的取消标记源，会发生什么情况：这个方法可能被多次调用，每次使用同一个（长寿命的）原有标记，这样每次都会连接一个新的标记源。即使 HTTP 请求已经结束了（没有用到组合标记），标记源仍会连在原有的标记上。为了防止这种内存泄漏的情况，一旦不再需要组合标记了，就要释放已连接的取消标记源。

参阅

9.9 节介绍发送取消请求的常规做法。

9.3 节介绍把取消用作超时。

9.9 与其他取消体系的互操作

问题

有一些外部的或以前遗留下来的代码采用了非标准的取消模式。现在要用标准的 CancellationToken 来控制这些代码。

解决方案

CancellationToken 类响应取消请求有两种主要的方式：轮询（见 9.2 节）和回调函数（本节的主题）。轮询通常用于计算密集型代码，如数据处理的循环。回调函数通常用于其他情况。可以用 CancellationToken.Register 方法注册一个取消标记的回调函数。

例如，假设我们要封装 System.Net.NetworkInformation.Ping 类，并且要实现对一个 ping 过程的取消。这个 Ping 类已经有基于 Task 的 API，但不支持 CancellationToken。但是 Ping 类有自己的 SendAsyncCancel 方法，可以用来取消一个 ping 过程。因此我们注册一个调用这个方法的回调函数，如下所示：

```
async Task<PingReply> PingAsync(string hostNameOrAddress,
    CancellationToken cancellationToken)
{
    var ping = new Ping();
    using (cancellationToken.Register(() => ping.SendAsyncCancel()))
    {
        return await ping.SendPingAsync(hostNameOrAddress);
    }
}
```

这样，在发出取消请求时这个回调函数就会调用 SendAsyncCancel 方法，来取消 SendPingAsync 的执行。

讨论

可以用 CancellationToken.Register 方法与任何类型的非主流取消体系进行互操作。但是有一点一定要记住，当一个方法使用 CancellationToken 后，一个取消请求应该只用来取消那一个操作。有些非主流的取消体系通过关闭一些资源来实现取消，而关闭资源会取消多个操作。此类取消体系就不大适合使用 CancellationToken。如果你一定要用 CancellationToken 封装此类取消体系，那就在文档中把这个不寻常的语法描述清楚吧。

要特别注意回调函数注册的生命周期。Register 方法返回一个可释放的对象，应该在不再需要回调函数时把它释放。前面的例子使用了 using 语句，会在异步操作结束时清理资源。如果不使用 using 语句，每次调用这个例子时使用同一个（长寿命的）CancellationToken，

它就会每次都添加一个回调函数（这个回调函数又会使 Ping 对象继续存活）。为了避免内存和资源的泄漏，一旦不再需要使用回调函数了，就要释放这个回调函数注册。

参阅

9.2 节介绍用轮询而不是用回调函数来响应取消标记。

9.1 节介绍用常规方法发出取消请求。

第10章

函数式OOP

现代程序需要异步编程，现在的服务器程序必须有更好的可扩展性，用户端程序必须有更好的交互性。开发者们意识到必须学习异步编程，他们在这个领域进行探索之后，发现异步编程经常会与已经习惯了的传统面向对象编程冲突。

冲突的主要原因在于异步编程是函数式的（functional）。这里“functional”的意思并不是“具备功能”，它是一种函数式编程方式，而不是过程式编程方式。很多开发人员在大学里学习了基本的函数式编程，后来却很少使用。如果像 `(car (cdr '(3 5 7)))` 这样的代码让你有似曾相识的感觉，那你属于这类人。但是不要害怕，只要你习惯了，现代的异步编程也没那么难。

引入 async 对异步开发的主要突破，是在异步编程时仍然可以用过程式编程的思维方式思考。这让异步方法的编写和理解变得更加容易。但是在内部实现中，异步代码本质上仍是函数式。在经典的面向对象设计中生硬地使用 async 方法，就会产生一些问题。本章讲述如何应对异步代码与面向对象编程发生的冲突。

在把已有的 OOP（面向对象编程）基础代码转换成 async 风格的基础代码时，这些冲突尤为明显。

10.1　异步接口和继承

问题

接口或基类中有一个方法，现在希望实现异步。

解决方案

理解本问题和解决方案的关键，是要知道异步是一种具体的实现方式。不可能把接口方法或抽象方法标记为 async。但是可以把一个方法编写得跟 async 方法一样，只不过不用 async 这个关键字。

要知道可以用 await 等待的是类，而不是方法。可以用 await 等待某个方法返回的 Task 对象，不管它是不是 async 方法。因此，一个接口或抽象方法可以返回一个 Task（或 Task<T>）对象，这个对象可以用 await 等待。

下面的代码定义了一个包含异步方法（不用 async）的接口、对该接口的实现（用 async），还定义了一个独立的方法，来调用该接口的方法（用 await）。

```
interface IMyAsyncInterface
{
    Task<int> CountBytesAsync(string url);
}

class MyAsyncClass : IMyAsyncInterface
{
    public async Task<int> CountBytesAsync(string url)
    {
        var client = new HttpClient();
        var bytes = await client.GetByteArrayAsync(url);
        return bytes.Length;
    }
}

static async Task UseMyInterfaceAsync(IMyAsyncInterface service)
{
    var result = await service.CountBytesAsync("http://www.example.com");
    Trace.WriteLine(result);
}
```

这种模式也适用于基类中的抽象方法。

异步方法的特征仅仅表示它的实现可以是异步的。如果没有真正的异步任务，用同步方式实现这个方法也是可以的。例如，在测试存根的代码中可以用 FromResult 来实现前面的接口（不用 async）：

```
class MyAsyncClassStub : IMyAsyncInterface
{
    public Task<int> CountBytesAsync(string url)
    {
        return Task.FromResult(13);
    }
}
```

讨论

在编写本书时（2014），async 和 await 推出还没多久。随着异步方法越来越普遍，在接口和基类上实现的异步方法也会越来越多。这其实并不难，只要记住两点：可以用 await 等待的是返回的类（而不是方法）；对一个异步方法的定义，可以用异步方式实现，也可以用同步方式实现。

参阅

2.2 节介绍用同步代码实现一个具有异步特征的方法，并返回一个已完成的 Task 对象。

10.2　异步构造：工厂

问题

需要在一个类的构造函数里进行异步操作。

解决方案

构造函数是不能异步的，也不能使用 await 关键字。假如能用 await 等待一个构造函数，当然能解决问题，但那需要对 C# 语言进行很大的改动。

一种可能是将构造函数与一个异步的初始化方法配对使用，就像下面这样使用类：

```
var instance = new MyAsyncClass();
await instance.InitializeAsync();
```

但是这种做法有缺点。很容易忘记调用 InitializeAsync 方法，并且类的实例在构造完后不能马上使用。

更好的解决方案，是为这个类建立自己的工厂。下面展示了异步工厂方法的模式：

```
class MyAsyncClass
{
    private MyAsyncClass()
    {
    }

    private async Task<MyAsyncClass> InitializeAsync()
    {
        await Task.Delay(TimeSpan.FromSeconds(1));
        return this;
    }

    public static Task<MyAsyncClass> CreateAsync()
```

```
    {
        var result = new MyAsyncClass();
        return result.InitializeAsync();
    }
}
```

构造函数和 InitializeAsync 是 private，因此其他代码不可能误用。创建实例的唯一方法是使用静态的 CreateAsyncfactory 方法，并且在初始化完成前，调用者是不能访问这个实例的。

其他代码可以这样创建一个实例：

```
var instance = await MyAsyncClass.CreateAsync();
```

讨论

这种模式的主要好处是，其他代码无法访问尚未初始化的 MyAsyncClass 实例。因此我建议大家尽可能采用这种模式，而不是其他方法。

可惜在有些情况下，这种方法无法使用，特别是当代码用到了依赖注入提供者的时候。在编写本书时（2014），主要的依赖注入或控制反转库都不能与异步代码一起使用。这种情况下，我们还有几个解决办法。

如果创建的实例是一个共享资源，那可以使用异步的 Lazy 类型（见 13.1 节）。否则，可以使用 10.3 节介绍的异步初始化模式。

请大家不要使用这样的代码：

```
class MyAsyncClass
{
    public MyAsyncClass()
    {
        InitializeAsync();
    }

    // 坏代码 !!
    private async void InitializeAsync()
    {
        await Task.Delay(TimeSpan.FromSeconds(1));
    }
}
```

这种做法初看起来好像很有道理：用一个常规的构造函数启动一个异步操作。但这样使用 async void 有几个缺点。第一个问题，当构造函数结束时，初始化过程仍在异步地进行，并且没有直观的方法可以监测异步的初始化过程是否已完成。第二个问题与错误处理有关：InitializeAsync 方法引发的任何错误，相关的 catch 语句都无法捕获。

参阅

10.3 节介绍异步初始化模式，针对使用了依赖注入 / 控制反转容器的情况，实现异步构造。

13.1 节介绍如何异步地初始化 Lazy 对象。如果实例在概念上是共享的资源或服务，这是一种可行的解决方案。

10.3 异步构造：异步初始化模式

问题

一个类的构造函数需要执行异步过程，但是不能使用异步工厂模式（见 10.2 节），因为这个类的实例是通过反射（如依赖注入 / 控制反转容器、数据绑定、`Activator.CreateInstance` 等）创建的。

解决方案

这种情况下必须返回一个未初始化的实例，但可以使用一种通用模式来减少不利因素：异步初始化模式。对于每个需要异步初始化的类，都要定义一个属性：

```
Task Initialization { get; }
```

我一般为需要异步初始化的类建一个标识接口（maker interface），在标识接口内定义这个属性：

```
/// <summary>
/// 把一个类标记为“需要异步初始化”
/// 并提供初始化的结果。
/// </summary>
public interface IAsyncInitialization
{
    /// <summary>
    /// 本实例的异步初始化的结果。
    /// </summary>
    Task Initialization { get; }
}
```

实现了这种模式后，就要在构造函数内启动初始化（并分配这个 `Initialization` 属性）。异步初始化的结果（包括所有的异常）是通过 `Initialization` 属性对外公开的。下面的例子实现了一个使用异步初始化的类：

```
class MyFundamentalType : IMyFundamentalType, IAsyncInitialization
{
    public MyFundamentalType()
    {
```

```
        Initialization = InitializeAsync();
    }

    public Task Initialization { get; private set; }

    private async Task InitializeAsync()
    {
        // 对这个实例进行异步初始化。
        await Task.Delay(TimeSpan.FromSeconds(1));
    }
}
```

如果使用了依赖注入/控制反转库，可以用下面的方式创建和初始化一个类的实例：

```
IMyFundamentalType instance = UltimateDIFactory.Create<IMyFundamentalType>();
var instanceAsyncInit = instance as IAsyncInitialization;
if (instanceAsyncInit != null)
    await instanceAsyncInit.Initialization;
```

可以对这种模式进行扩展，将类和异步初始化结合起来。下面的例子定义了另一个类，它以前面建立的 IMyFundamentalType 为基础：

```
class MyComposedType : IMyComposedType, IAsyncInitialization
{
    private readonly IMyFundamentalType _fundamental;

    public MyComposedType(IMyFundamentalType fundamental)
    {
        _fundamental = fundamental;
        Initialization = InitializeAsync();
    }

    public Task Initialization { get; private set; }

    private async Task InitializeAsync()
    {
        // 如有必要，异步地等待基础实例的初始化。
        var fundamentalAsyncInit = _fundamental as IAsyncInitialization;
        if (fundamentalAsyncInit != null)
            await fundamentalAsyncInit.Initialization;

        // 做自己的初始化工作（同步或异步）。
        ...
    }
}
```

这个混合类在进行它自己的初始化之前，先等待它的所有部件都初始化完毕。需要遵循一个规则，即在 InitializeAsync 结束前，每个部件都必须初始化完毕。只要混合类的初始化过程完成了，就能保证它所依赖的每个类型也是经过初始化的。部件的初始化过程中产生的任何异常，会传递给混合类的初始化过程。

讨论

建议大家尽量不要使用这个解决方案，而是使用异步工厂（见 10.2 节）或异步 Lazy 对象初始化（见 13.1 节）。那些才是最好的方法，因为它们绝不可能暴露未初始化的实例。但是，如果用注入依赖 / 控制反转、数据绑定方式等创建实例，那就不可避免地要暴露未初始化的实例。这种情况下，就推荐使用本节讲述的异步初始化模式。

从异步接口（见 10.1 节）开始我们就讲过，异步方法的特征仅仅表示这个方法可以是异步的。前面的 MyComposedType.InitializeAsync 代码正好说明这点：如果 IMyFundamentalType 实例也没有实现 IAsyncInitialization，并且 MyComposedType 没有用异步方式对自己进行初始化，那么它的 InitializeAsync 方法实际上会同步地完成。

检查某个实例是否实现了 IAsyncInitialization，并对它做初始化，这个过程的代码非常臃肿，尤其是这个组合类有很多部件的情况。有一个很容易的办法可以简化代码，就是创建一个辅助方法：

```
public static class AsyncInitialization
{
    static Task WhenAllInitializedAsync(params object[] instances)
    {
        return Task.WhenAll(instances
            .OfType<IAsyncInitialization>()
            .Select(x => x.Initialization));
    }
}
```

可以调用 InitializeAllAsync 并传入需要初始化的任何实例。这个方法会忽略那些未实现 IAsyncInitialization 的实例。如果一个组合类依赖了三个注入的实例，它的初始化代码就可以这么写：

```
private async Task InitializeAsync()
{
    // 异步地等待三个实例全部初始化完毕（有些可能不需要初始化）。
    await AsyncInitialization.WhenAllInitializedAsync(_fundamental,
        _anotherType, _yetAnother);

    // 做自己的初始化工作（同步或异步）。
    ...
}
```

参阅

10.2 节介绍了异步工厂，用它可以异步地构造实例，而不会公开未初始化的实例。

13.1 节介绍异步地初试化 Lazy 对象，用于实例为共享资源或服务的情况。

10.1 节介绍异步接口。

10.4 异步属性

问题

要把一个属性改成异步方式。该属性不会用于数据绑定。

解决方案

在使用 async 改造原有代码时，经常会出现这样的问题。这时你会在属性的 get 方法中调用一个异步方法。但实际上根本没有“异步属性”这种东西。不允许在属性中使用 async 关键字，这么规定是有好处的。属性的 get 方法应该返回当前值，不应该启动一个后台的操作：

```
// 我们假想的代码（请不要编译）。
public int Data
{
    async get
    {
        await Task.Delay(TimeSpan.FromSeconds(1));
        return 13;
    }
}
```

当你发现需要在代码中加入一个“异步属性”时，实际上应该选用其他方案。选用哪种解决方案取决于要对属性值计算一次还是多次。需要确定是下面两种情况中的哪一种：

- 每次读取属性时，都要对值进行异步计算；
- 异步地计算属性值一次，并缓存起来供以后访问。

如果每次读取“异步属性”时都要启动一次新的（异步的）计算过程，那说明它不是一个*属性*。它实际上是一个经过伪装的*方法*。如果你在把同步代码转换成异步代码的时候遇到这种情况，那就要意识到原始的设计就是错误的。实际上这个属性从一开始就应该是一个方法：

```
// 作为一个异步方法。
public async Task<int> GetDataAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    return 13;
}
```

属性也可以直接返回一个 Task<int>，例如：

```
// 作为一个返回 Task 的属性。
// 这个 API 设计是有问题的。
public Task<int> Data
{
    get { return GetDataAsync(); }
}
```

```
private async Task<int> GetDataAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    return 13;
}
```

但是我建议大家不要采用这种做法。如果每次访问属性都会启动一次新的异步操作，那说明这个“属性”其实应该是一个方法。当它是异步的方法，人们就会更清楚地知道每次访问都会开始新的异步操作，因此这个 API 就不会误导别人。10.3 节和 10.6 节中确实有返回 Task 的属性，但它们是作为一个整体被实例使用的。每次读取它们时不会启动新的异步操作。

前面的解决方案用于每次访问都会计算属性值的情况。另一种情况是“异步属性”只启动一次（异步）计算，并缓存计算结果供以后使用。针对这种情况，可以使用异步的 Lazy 对象初始化。这方面将在 13.1 节详述，现在先来看一下代码：

```
// 作为一个缓存的数据。
public AsyncLazy<int> Data
{
    get { return _data; }
}

private readonly AsyncLazy<int> _data =
    new AsyncLazy<int>(async () =>
    {
        await Task.Delay(TimeSpan.FromSeconds(1));
        return 13;
    });
```

这段代码只会异步地计算一次，然后每次向调用者返回同一个值。调用的代码如下：

```
int value = await instance.Data;
```

这时，计算过程只会发生一次，因此使用属性是合适的。

讨论

有一个重要的问题需要明确：读取属性时是否要启动一个新的异步操作。如果答案为“是”，那就不要用属性，改用异步方法。如果属性要充当一个惰性求值（lazy-evaluated）的缓存，那就使用异步初始化（见 13.1 节）。13.3 节会介绍用于数据绑定的属性。

在把同步属性转换为“异步属性”时，不能这么做：

```
private async Task<int> GetDataAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    return 13;
}
```

```
public int Data
{
    // 坏代码！！
    get { return GetDataAsync().Result; }
}
```

不要用 Result 或 Wait 把异步代码强制转换为同步代码。在 GUI 和 ASP.NET 平台中，这样的代码很容易造成死锁。即使绕过了死锁，这也是一个容易引起误解的 API：一个属性的 get 方法（它本该是一个快速、同步的操作）其实是一个阻塞的操作。关于这种阻塞问题，第 1 章有更详细的描述。

既然我们在讨论异步代码中的属性，那就有必要考虑一下状态与异步代码的关系。在把同步的基础代码转换成异步代码时，这一点尤为重要。看一下 API 对外提供的每个状态（例如通过属性提供）。思考一个问题：在异步操作进行的过程中，该对象的当前状态是什么？这个问题没有正确的答案，但要考虑清楚你想要表达的语义，并把它写进文档，这一点很重要。

举个例子，来看 Stream.Position，它表示一个流的指针当前偏移位置。使用同步 API 的情况，当调用 Stream.Read 或 Stream.Write 时，它就完成了实际的读 / 写功能，并且在 Read 或 Write 返回前更新 Stream.Position 的位置。这些同步代码的语义是非常明确的。

现在来考虑 Stream.ReadAsync 和 Stream.WriteAsync 的情况：什么时候修改 Stream.Position？在读 / 写操作结束时，还是在真正开始读 / 写之前？如果是在结束前修改，那是在 ReadAsync/WriteAsync 返回时同步地修改，还是在返回后立即修改？

这个例子很好地说明了：对于同步代码来说，对外提供状态的属性其语义是非常明确的，但对于异步代码就没有明显正确的语义了。当然事情没那么可怕：在把类改成异步方式时，只需要对整个 API 进行全局考虑，并把选定的语义在文档中描述清楚。

参阅

13.1 节详细介绍了异步 Lazy 对象初始化。

13.3 节介绍了需要支持数据绑定的“异步属性”。

10.5　异步事件

问题

需要一个可以异步运行的事件处理器，并且需要监测事件处理器是否已经完成。注意这种情况非常少见。一般来说，提出一个事件后就不再关心事件处理器何时完成。

解决方案

监测 async void 类型的事件处理器什么时候返回是根本不可行的，因此需要采用其他办法。Windows 应用商店平台引入了一个名为延期（deferral）的概念，可用来跟踪异步事件处理器。异步事件处理器在第一次使用 await 前分配一个延期对象，处理结束时通知这个延期对象。同步的事件处理器不需要使用延期。

Nito.AsyncEx 库中有一个 DeferralManager 类（延期管理器），引发事件的组件可使用它。事件处理器可以利用这个延期管理器来分配延期对象，并且跟踪每个延期对象的完成时间。

对每个需要等待处理完毕的事件，首先要扩展事件参数类：

```
public class MyEventArgs : EventArgs
{
    private readonly DeferralManager _deferrals = new DeferralManager();

    ... // 自身的构造函数和属性。

    public IDisposable GetDeferral()
    {
        return _deferrals.GetDeferral();
    }

    internal Task WaitForDeferralsAsync()
    {
        return _deferrals.SignalAndWaitAsync();
    }
}
```

在编写异步事件处理器时，事件参数类最好是线程安全的。要做到这点，最简单的办法就是让它成为不可变的（即把所有的属性都设为只读）。

接着，就可以在每次引发事件后（异步地）等待，直到所有异步事件处理器完成。如果没有事件处理器，下面的代码会返回一个已完成的 Task 对象；否则会创建一个事件参数类的新实例，并传入事件处理器，然后等待任意异步事件处理器的完成：

```
public event EventHandler<MyEventArgs> MyEvent;

private Task RaiseMyEventAsync()
{
    var handler = MyEvent;
    if (handler == null)
        return Task.FromResult(0);

    var args = new MyEventArgs(...);
    handler(this, args);
    return args.WaitForDeferralsAsync();
}
```

然后异步事件处理器可以在 using 块中使用延期对象，该延期对象会在被销毁时通知延期管理器：

```
async void AsyncHandler(object sender, MyEventArgs args)
{
    using (args.GetDeferral())
    {
        await Task.Delay(TimeSpan.FromSeconds(2));
    }
}
```

这种方式与 Windows 应用商店平台的延期对象有细微的差别。在 Windows 应用商店 API 中，每个需要延期对象的事件定义自己的延期类。而且这个延期类有一个显式定义的 Complete 方法，而不是 IDisposable。

讨论

.NET 中的事件在逻辑上可以分为两类，它们之间的语义差别很大。为了区分，我称它们为*通知事件*和*命令事件*。这不是官方的术语，仅仅是我为了便于区分而选用的名词。引发一个通知事件是为了把一些情况通知给其他部件。通知完全是单向的，事件的发送者并不关心有没有事件的接收者。处理通知时，发送者和接收者可以是彻底分离的。大部分事件属于通知事件，例如鼠标点击事件。

相反，引发一个命令事件是因为发送消息的部件想要实现一些功能。虽然经常把命令事件作为 .NET 事件处理，但命令事件其实并不是“事件”，它不符合“事件”这个词的本意。命令发送者在继续运行之前，必须等待接收者处理完事件。用来实现访问者模式（Visitor pattern）的事件，这就是命令事件。生命周期事件也是命令事件，因此 ASP.NET 页面的生命周期事件和 Windows 应用商店事件（例如 Application.Suspending）都属于这类。本质上是一个实现过程的事件，也都是命令事件（例如 BackgroundWorker.DoWork）。

通知事件并不需要任何特殊代码就可以使用异步的事件处理器。事件处理器可以是 async void 类型的，运行起来不会有任何问题。当事件发送者引发某个事件，异步的事件处理器不会立即完成。但是这并不影响什么，因为它们只是通知事件。对于通知事件，为了支持异步的事件处理器一共需要做多少工作？答案是：什么也不用做。

命令事件的情况就不同了。对于命令事件，就必须有一种方法来监测事件处理器何时完成。前面使用延期对象的解决方案，只适用于命令事件。



DeferralManager 类在 NuGet 包 Nito.AsyncEx 中。

参阅

第 2 章介绍了异步编程的基础知识。

10.6　异步销毁

问题

已经有了一个可以进行异步操作的类，现在需要能够释放它的资源。

解决方案

在销毁一个实例时，有几种方法来处理正在执行的操作：可以把销毁看做是一个针对所有正在运行的操作的取消请求，或者实现一个真正的*异步完成*。

把销毁看成一个取消请求是 Windows 平台上的惯例。文件流和套接字在关闭时，会取消所有运行中的读 / 写过程。在 .NET 环境下也可采用类似的做法，定义一个私有的 `CancellationTokenSource` 对象，并把取消标记传递给内部的操作。用下面的代码，`Dispose` 会取消这些操作，但不会等待操作的完成：

```
class MyClass : IDisposable
{
    private readonly CancellationTokenSource _disposeCts =
        new CancellationTokenSource();

    public async Task<int> CalculateValueAsync()
    {
        await Task.Delay(TimeSpan.FromSeconds(2), _disposeCts.Token);
        return 13;
    }

    public void Dispose()
    {
        _disposeCts.Cancel();
    }
}
```

前面的代码展示了有关 `Dispose` 的基本模式。在实际开发中应该增加一项检查，以确认对象还没有被销毁，还要支持方法本身的 `CancellationToken`（使用 9.8 节的技术）：

```
public async Task<int> CalculateValueAsync(CancellationToken cancellationToken)
{
    using (var combinedCts = CancellationTokenSource
        .CreateLinkedTokenSource(cancellationToken, _disposeCts.Token))
    {
        await Task.Delay(TimeSpan.FromSeconds(2), combinedCts.Token);
        return 13;
    }
}
```

当 Dispose 被调用时，调用程序中所有运行中的操作都会被取消：

```
async Task Test()
{
    Task<int> task;
    using (var resource = new MyClass())
    {
        task = CalculateValueAsync();
    }

    // 抛出异常 OperationCanceledException.
    var result = await task;
}
```

在 Dispose 的实现中生成一个取消请求，这种方法对于有些类运行得很好（例如 Http-Client 就有这种语法）。但是其他一些类需要知道操作完成的时间。对这些类就需要采用实现“异步完成”的方式了。

异步完成与异步初始化（见 10.3 节）很相似：它们都很少有官方的指引资料。因此本书介绍一种可行的模式，它基于 TPL 数据流块的运行方式。异步完成的重要部分可以封装在一个接口中：

```
/// <summary>
/// 表明一个类需要异步完成，并提供完成的结果。
/// </summary>
interface IAsyncCompletion
{
    /// <summary>
    /// 开始本实例的完成过程。概念上类似于“IDisposable.Dispose”。
    /// 在调用本方法后，就不能调用除了“Completion”以外的任何成员。
    /// </summary>
    void Complete();

    /// <summary>
    /// 取得本实例完成的结果。
    /// </summary>
    Task Completion { get; }
}
```

实现的类可用如下代码：

```
class MyClass : IAsyncCompletion
{
    private readonly TaskCompletionSource<object> _completion =
        new TaskCompletionSource<object>();
    private Task _completing;

    public Task Completion
    {
        get { return _completion.Task; }
    }
```

```
    public void Complete()
    {
        if (_completing != null)
            return;
        _completing = CompleteAsync();
    }

    private async Task CompleteAsync()
    {
        try
        {
            ... // 异步地等待任何运行中的操作。
        }
        catch (Exception ex)
        {
            _completion.TrySetException(ex);
        }
        finally
        {
            _completion.TrySetResult(null);
        }
    }
}
```

调用它的代码看起来不那么漂亮，Dispose 必须是异步的，因此不能使用 using 语句。但是我们可以定义一对辅助方法，完成类似 using 语句的功能：

```
static class AsyncHelpers
{
    public static async Task Using<TResource>(Func<TResource> construct,
        Func<TResource, Task> process) where TResource : IAsyncCompletion
    {
        // 创建需要使用的资源。
        var resource = construct();

        // 使用资源，并捕获所有异常。
        Exception exception = null;
        try
        {
            await process(resource);
        }
        catch (Exception ex)
        {
            exception = ex;
        }

        // 完成（逻辑上销毁）资源。
        resource.Complete();
        await resource.Completion;

        // 如果需要，就重新抛出“process”产生的异常。
        if (exception != null)
            ExceptionDispatchInfo.Capture(exception).Throw();
```

```
    }

    public static async Task<TResult> Using<TResource, TResult>(
        Func<TResource> construct, Func<TResource,
        Task<TResult>> process) where TResource : IAsyncCompletion
    {
        // 创建需要使用的资源。
        var resource = construct();

        // 使用资源，并捕获所有异常。
        Exception exception = null;
        TResult result = default(TResult);
        try
        {
            result = await process(resource);
        }
        catch (Exception ex)
        {
            exception = ex;
        }

        // 完成（逻辑上销毁）资源。
        resource.Complete();
        try
        {
            await resource.Completion;
        }
        catch
        {
            // 只有当“process”没有抛出异常时，才允许抛出“Completion”的异常。
            if (exception == null)
                throw;
        }

        // 如果需要，就重新抛出“process”产生的异常。
        if (exception != null)
            ExceptionDispatchInfo.Capture(exception).Throw();

        return result;
    }
}
```

代码中使用了 ExceptionDispatchInfo，以保留异常的栈轨迹。准备好这些辅助方法后，调用的代码就可以这样使用 Using 方法了：

```
async Task Test()
{
    await AsyncHelpers.Using(() => new MyClass(), async resource =>
    {
        // 使用资源。
    });
}
```

讨论

跟用 Dispose 实现取消请求的方式相比，异步完成的方式显然要麻烦得多，只有在确实需要时才能使用这种方法。其实在大多数情况下是不需要销毁任何东西的，这当然是最简单的方法，因为什么都不需要做。

本节介绍的异步完成模式，用在 TPL 数据流块和少数其他类中（例如 ConcurrentExclusiveSchedulerPair）。数据流块还有一种完成请求类型，表示它在结束时会产生错误 (IDataflowBlock.Fault(Exception))。在自定义的类中也可以使用这种模式，因此可把本节中的 IAsyncCompletion 作为一个如何实现异步完成的例子。

本节介绍了两种处理销毁过程的模式，这两种模式也可以同时使用。一个类同时使用这两种模式后，当客户端代码使用 Complete 和 Completion，这个类就会正常地关闭；当客户端代码使用 Dispose，这个类就会“取消”操作。

参阅

10.3 节介绍异步初始化模式。

MSDN 中关于 TPL 数据流的文档，介绍数据流块的完成和正常关闭。

9.8 节介绍互相连接的取消标记。

10.1 节介绍异步接口。

第11章

同步

如果程序用到了并发技术（几乎所有 .NET 程序都用了），那就要特别留意这种情况：一段代码需要修改数据，同时其他代码需要访问同一个数据。这种情况出现时，就需要*同步地*访问数据。本章的各小节介绍了用于同步访问的最常用的类。其实从本书中的其他章节中可以看出，一些程序库本身就已经做了很多更普遍的同步工作。在详细介绍同步技术分类之前，我们先来仔细地看一下一些常见的、需要使用或不需要使用同步的情况。



本段内容对同步的解释做了一定的简化，但是这些结论都是正确的。

同步的类型主要有两种：*通信*和*数据保护*。当一段代码把某些情况（例如收到新消息）通知给另一段代码时，就得用到通信。在后面的有关案例中会详细讲述通信。本章的概述部分主要讨论数据保护。

如果下面三个条件都满足，就需要用同步来保护共享的数据。

- 多段代码正在并发运行。
- 这几段代码在访问（读或写）同一个数据。
- 至少有一段代码在修改（写）数据。

第一个条件的原因很容易理解。如果整个代码只是从头到尾地运行，没有任何并发，那就根本不用担心同步问题。有些简单的控制台程序是这样的，但是绝大多数 .NET 程序肯定

用到了某些类型的并发功能。第二个条件是说如果每段代码都有自己非共享的局部数据，那就不需要同步。局部数据是独立于其他代码的。如果有共享数据，但数据永远不会修改的话，那也没必要使用同步。第三个条件的情况包括，在程序启动时一旦设置了配置参数，就永不修改。如果共享的数据只是用来读取的，就不需要同步。

数据保护是为了每段代码访问数据时能得到一致的结果。一段代码正在修改数据时要使用同步技术，以保证在系统的其他部分看来这些修改具有原子性。

只有经过实践才会知道什么时候需要同步，因此在开始讨论具体方法之前，我们先看几个例子。这是第一个例子：

```
async Task MyMethodAsync()
{
    int val = 10;
    await Task.Delay(TimeSpan.FromSeconds(1));
    val = val + 1;
    await Task.Delay(TimeSpan.FromSeconds(1));
    val = val - 1;
    await Task.Delay(TimeSpan.FromSeconds(1));
    Trace.WriteLine(val);
}
```

如果从一个线程池线程调用这个方法（例如在 `Task.Run` 中运行），访问 `val` 的代码行会在独立的线程池线程中运行。但它需要同步吗？不，因为这些代码行不会同时运行。这个方法是异步的，但是它也是按顺序运行的（一次只会处理一部分）。

好，我们把例子改复杂一些。这次来运行并发异步的代码：

```
class SharedData
{
    public int Value { get; set; }
}

async Task ModifyValueAsync(SharedData data)
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    data.Value = data.Value + 1;
}

// 警告：可能需要同步，见下面的讨论。
async Task<int> ModifyValueConcurrentlyAsync()
{
    var data = new SharedData();

    // 启动三个并发的修改过程。
    var task1 = ModifyValueAsync(data);
    var task2 = ModifyValueAsync(data);
    var task3 = ModifyValueAsync(data);

    await Task.WhenAll(task1, task2, task3);
```

```
        return data.Value;
    }
```

本例中，启动了三个并发运行的修改过程。需要同步吗？答案是“看情况”。如果能确定这个方法是在 GUI 或 ASP.NET 上下文中调用的（或同一时间内只允许一段代码运行的任何其他上下文），那就不需要同步，因为这三个修改数据过程的运行时间是互不相同的。例如，如果它在 GUI 上下文中运行，就只有一个 UI 线程可以运行这些数据修改过程，因此一段时间内只能运行一个过程。因此，如果能够确定是“同一时间只运行一段代码”的上下文，那就不需要同步。但是如果从线程池线程（如 Task.Run）调用这个方法，就需要同步了。在那种情况下，这三个数据修改过程会在独立的线程池线程中运行，并且同时修改 data.Value，因此必须同步地访问 data.Value。

现在我们把数据改为私有成员，代替需要传递的变量，并且来看一个新的技巧：

```
    private int value;

    async Task ModifyValueAsync()
    {
        await Task.Delay(TimeSpan.FromSeconds(1));
        value = value + 1;
    }

    // 警告：可能需要同步，见下面的讨论。
    async Task<int> ModifyValueConcurrentlyAsync()
    {
        // 启动三个并发的修改过程。
        var task1 = ModifyValueAsync();
        var task2 = ModifyValueAsync();
        var task3 = ModifyValueAsync();

        await Task.WhenAll(task1, task2, task3);

        return value;
    }
```

上面讨论的内容对这段代码也同样适用。如果调用这个方法的是线程池上下文，那显然需要同步。但这里还有一个技巧。前一个例子创建了在三个修改方法中共享的 SharedData 实例。这个例子则用一个具体的私有成员作为共享数据。这意味着如果 ModifyValueConcurrentlyAsync 被多次调用，每个独立的调用都会共享这个 value。如果要避免这种共享，即使在“同一时间只运行一段代码”上下文中，也需要使用同步。换句话说，如果要在每次调用 ModifyValueConcurrentlyAsync 之前等待前面的调用完成，那就需要加入同步。即使上下文能够确保只有一个线程来运行所有代码（即 UI 线程），也是如此。这种情况下的同步，实际上是对异步方法的一种限流（见 11.2 节）。

我们再来看一个异步例子。可以用 Task.Run 来做“简单的并行”——一种基本的并行处理，不像真正的 Parallel/PLINQ 并行那样考虑效率和可配置性。下面的代码用“简单的

并行”修改一个共享数据：

```
// 坏代码！！
async Task<int> SimpleParallelismAsync()
{
    int val = 0;
    var task1 = Task.Run(() => { val = val + 1; });
    var task2 = Task.Run(() => { val = val + 1; });
    var task3 = Task.Run(() => { val = val + 1; });
    await Task.WhenAll(task1, task2, task3);
    return val;
}
```

线程池中运行了三个独立的任务（通过 Task.Run），都在修改同一个变量 val。这满足了前面讲的几个条件，毫无疑问需要同步。注意，即使 val 是一个局部变量，这段代码也需要同步。虽然它是一个方法内的局部变量，但仍被多个线程共享。

可把上面的代码改造成真正的并行代码，我们来看使用 Parallel 类的例子：

```
void IndependentParallelism(IEnumerable<int> values)
{
    Parallel.ForEach(values, item => Trace.WriteLine(item));
}
```

既然用到了 Parallel 类，那肯定有多个线程。但是并行的循环体（item => Trace.WriteLine(item)）只是读取它自己的数据。这里没有在多个线程间共享的数据。Parallel 类把数据分配给各个线程，因此它们没必要共享数据。每个线程运行自己的循环体，且独立于运行相同循环体的其他线程。因此这段代码不需要同步。

我们来看一个聚合的例子，与 3.2 节中的一个例子类似：

```
// 坏代码！！
int ParallelSum(IEnumerable<int> values)
{
    int result = 0;
    Parallel.ForEach(source: values,
        localInit: () => 0,
        body: (item, state, localValue) => localValue + item,
        localFinally: localValue => { result += localValue; });
    return result;
}
```

这个例子也使用了多线程，这次每个线程在启动时把局部变量初始化为 0（() => 0）。线程对每个输入值的处理，是把输入值累加到它的局部变量（(item, state, localValue) => localValue + item）。最后所有的局部变量被累加到返回值（localValue=> { result += localValue; }）。前面两步没有问题，因为线程间还没有共享数据。局部变量和输入数据在各个线程之间是互相独立的。但是最后一个步骤就有问题了。当每个线程都把它的局部变量累加到返回值时，就出现了多个线程访问并修改同一个共享变量（result）的情况。

因此最后一个步骤需要同步（见 11.1 节）。

PLINQ、数据流、响应式编程库的情况与 Parallel 非常类似：只要代码只处理它自己的输入数据，就不需要考虑同步问题。我发现只要正确地使用这些库，大多数代码都几乎不用增加同步功能。

最后我们来看一下集合。记住需要同步的三个条件是：*多段代码、共享数据、修改数据*。

不可变类型本身就是线程安全的，因为它们是不会改变的。修改一个不可变集合是不可能的，因此根本不需要同步。例如下面的代码不需要同步，因为每个独立的线程池线程向栈压入一个值时，实际上是用这个值创建了一个新的不可变栈，而原始栈保持不变：

```
async Task<bool> PlayWithStackAsync()
{
    var stack = ImmutableStack<int>.Empty;

    var task1 = Task.Run(() => Trace.WriteLine(stack.Push(3).Peek()));
    var task2 = Task.Run(() => Trace.WriteLine(stack.Push(5).Peek()));
    var task3 = Task.Run(() => Trace.WriteLine(stack.Push(7).Peek()));
    await Task.WhenAll(task1, task2, task3);

    return stack.IsEmpty; // 总是返回 true。
}
```

但是，在使用不可变集合时通常会有一个共享的“根”变量，它本身不是不可变的。这时就必须使用同步了。下面的代码中每个线程向栈压入一个值（同时创建一个新的不可变栈），然后修改这个共享的根变量。在这个例子中，修改这个栈变量时是需要使用同步的：

```
// 坏代码！！
async Task<bool> PlayWithStackAsync()
{
    var stack = ImmutableStack<int>.Empty;

    var task1 = Task.Run(() => { stack = stack.Push(3); });
    var task2 = Task.Run(() => { stack = stack.Push(5); });
    var task3 = Task.Run(() => { stack = stack.Push(7); });
    await Task.WhenAll(task1, task2, task3);

    return stack.IsEmpty;
}
```

线程安全集合（例如 ConcurrentDictionary）就完全不同了。与不可变集合不同，线程安全集合是可以修改的。线程安全集合本身就包含了所有的同步功能，因此根本不需要担心同步的问题。下面的代码，如果修改的是 Dictionary 而不是 ConcurrentDictionary，那就需要同步。但事实上它是在修改 ConcurrentDictionary，因此就不需要同步：

```
async Task<int> ThreadsafeCollectionsAsync()
{
```

```
    var dictionary = new ConcurrentDictionary<int, int>();

    var task1 = Task.Run(() => { dictionary.TryAdd(2, 3); });
    var task2 = Task.Run(() => { dictionary.TryAdd(3, 5); });
    var task3 = Task.Run(() => { dictionary.TryAdd(5, 7); });
    await Task.WhenAll(task1, task2, task3);

    return dictionary.Count; // 总是返回 3。
}
```

11.1 阻塞锁

问题

多个线程需要安全地读写共享数据。

决解方案

这种情况最好的办法是使用 `lock` 语句。一个线程进入锁后，在锁被释放之前其他线程是无法进入的：

```
class MyClass
{
    // 这个锁会保护 _value。
    private readonly object _mutex = new object();

    private int _value;

    public void Increment()
    {
        lock (_mutex)
        {
            _value = _value + 1;
        }
    }
}
```

讨论

.NET 框架中还有很多其他类型的锁，如 `Monitor`、`SpinLock`、`ReaderWriterLockSlim`。对大多数程序来说，这些类型的锁基本上用不到。尤其是程序员会习惯性地使用 `ReaderWriterLockSlim`，即使没必要用那么复杂的技术。基本的 `lock` 语句就可以很好地处理 99% 的情况了。

关于锁的使用，有四条重要的规则。

- 限制锁的作用范围。

- 文档中写清锁保护的内容。
- 锁范围内的代码尽量少。
- 在控制锁的时候绝不运行随意的代码。

首先，要尽量限制锁的作用范围。应该把 lock 语句使用的对象设为私有成员，并且永远不要暴露给非本类的方法。每个类型通常最多只有一个锁。如果一个类型有多个锁，可考虑通过重构把它分拆成多个独立的类型。可以锁定任何引用类型，但是我建议为 lock 语句定义一个专用的成员，就像最后的例子那样。尤其是千万不要用 lock(this)，也不要锁定 Type 或 string 类型的实例。因为这些对象是可以被其他代码访问的，这样锁定会产生死锁。

第二，要在文档中描述锁定的内容。这种做法在最初编写代码时很容易被忽略，但是在代码变得复杂后就会变得很重要。

第三，在锁定时执行的代码要尽可能得少。要特别小心阻塞调用。在锁定时不要做任何阻塞操作。

最后，在锁定时绝不要调用随意的代码。随意的代码包括引发事件、调用虚拟方法、调用委托。如果一定要运行随意的代码，就在释放锁之后运行。

参阅

11.2 节介绍兼容 async 的锁。本节介绍的 lock 语句与 await 并不兼容。

11.3 节介绍线程间的信号。本节介绍的 lock 语句是用来保护共享数据的，而不是在线程间发送信号。

11.5 节介绍限流，它扩大了锁的概念。一个锁可以理解为每次只允许一个的限流。

11.2 异步锁

问题

多个代码块需要安全地读写共享数据，并且这些代码块可能使用 await 语句。

解决方案

.NET 4.5 对框架中的 SemaphoreSlim 类进行了升级以兼容 async。可以这样使用：

```
class MyClass
{
    // 这个锁保护 _value。
    private readonly SemaphoreSlim _mutex = new SemaphoreSlim(1);
```

```
    private int _value;

    public async Task DelayAndIncrementAsync()
    {
        await _mutex.WaitAsync();
        try
        {
            var oldValue = _value;
            await Task.Delay(TimeSpan.FromSeconds(oldValue));
            _value = oldValue + 1;
        }
        finally
        {
            _mutex.Release();
        }
    }
}
```

不过，只有在 .NET 4.5 或更高版本中才能以这种方式使用 `SemaphoreSlim`。如果使用的是旧版本框架或者是在编写可移植类库，那可以使用 `Nito.AsyncEx` 库中的 `AsyncLock` 类：

```
class MyClass
{
    // 这个锁保护 _value。
    private readonly AsyncLock _mutex = new AsyncLock();

    private int _value;

    public async Task DelayAndIncrementAsync()
    {
        using (await _mutex.LockAsync())
        {
            var oldValue = _value;
            await Task.Delay(TimeSpan.FromSeconds(oldValue));
            _value = oldValue + 1;
        }
    }
}
```

讨论

11.1 节中的规则在这里也同样适用，包括：

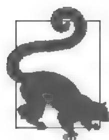
- 限制锁的作用范围；
- 文档中写清锁保护的内容；
- 锁范围内的代码尽量少；
- 在控制锁的时候绝不运行随意的代码。

确保锁的实例是私有的。不要暴露到类的外面。确保文档清晰（同时要做全面仔细的考虑），准确地描述锁保护的内容。在控制锁时执行的代码要尽量少。尤其是不要运行随意的代码，

包括引发事件、调用虚拟方法、以及调用委托。各平台对异步锁的支持情况见表 11-1。

表11-1：各平台对异步锁的支持

平　台	`SemaphoreSlim.WaitAsync`	`AsyncLock`
.NET 4.5	√	√
.NET 4.0	×	√
Mono iOS/Droid	√	√
Windows Store	√	√
Windows Phone Apps 8.1	√	√
Windows Phone SL 8.0	√	√
Windows Phone SL 7.1	×	√
Silverlight 5	×	√



`AsyncLock` 类型在 NuGet 包 `Nito.AsyncEx` 中。

参阅

11.4 节介绍了兼容 `async` 的信号。锁是用来保护共享数据的，不能作为信号。

11.5 节介绍限流，它扩大了锁的概念。一个锁可以理解为每次只允许一个的限流。

11.3　阻塞信号

问题

需要从一个线程发送信号给另一个线程。

解决方案

最常见和通用的跨线程信号是 `ManualResetEventSlim`。一个人工重置的事件处于这两种状态其中之一：标记的（signaled）或未标记的（unsignaled）。每个线程都可以把事件设置为 signaled 状态，也可以把它重置为 unsignaled 状态。线程也可等待事件变为 signaled 状态。

下面的两个方法被两个独立的线程调用，一个线程等待另一个线程的信号：

```
class MyClass
{
    private readonly ManualResetEventSlim _initialized =
```

```
            new ManualResetEventSlim();

        private int _value;

        public int WaitForInitialization()
        {
            _initialized.Wait();
            return _value;
        }

        public void InitializeFromAnotherThread()
        {
            _value = 13;
            _initialized.Set();
        }
    }
```

讨论

ManualResetEventSlim 是功能强大、通用的线程间信号，但必须合理地使用。如果这个信号其实是一个线程间发送小块数据的消息，那可考虑使用生产者 / 消费者队列。另一方面，如果信号只是用来协调对共享数据的访问，那可改用锁。

在 .NET 框架中，还有一些不常用的线程同步信号类型。如果 ManualResetEventSlim 不能满足需求，还可考虑用 AutoResetEvent、CountdownEvent 或 Barrier。

参阅

8.6 节介绍了阻塞的生产者 / 消费者队列。

11.1 节介绍了阻塞锁。

11.4 节介绍了兼容 async 的信号。

11.4　异步信号

问题

需要在代码的各个部分间发送通知，并且要求接收方必须进行异步等待。

解决方案

如果该通知只需要发送一次，那可用 TaskCompletionSource<T> 异步发送。发送代码调用 TrySetResult，接收代码等待它的 Task 属性：

```
class MyClass
{
    private readonly TaskCompletionSource<object> _initialized =
        new TaskCompletionSource<object>();

    private int _value1;
    private int _value2;

    public async Task<int> WaitForInitializationAsync()
    {
        await _initialized.Task;
        return _value1 + _value2;
    }

    public void Initialize()
    {
        _value1 = 13;
        _value2 = 17;
        _initialized.TrySetResult(null);
    }
}
```

在所有情况下都可以用 TaskCompletionSource<T> 来异步地等待：本例中，通知来自于另一部分代码。如果只需要发送一次信号，这种方法很适合。但是如果要打开和关闭信号，这种方法就不大合适了。

Nito.AsyncEx 库中有一个 AsyncManualResetEvent 类，基本上相当于是异步的 ManualResetEvent。下面是一个虚拟的例子，但说明了如何使用 AsyncManualResetEvent 类：

```
class MyClass
{
    private readonly AsyncManualResetEvent _connected =
        new AsyncManualResetEvent();

    public async Task WaitForConnectedAsync()
    {
        await _connected.WaitAsync();
    }

    public void ConnectedChanged(bool connected)
    {
        if (connected)
            _connected.Set();
        else
            _connected.Reset();
    }
}
```

讨论

信号是一种通用的通知机制。但如果这个“信号”是一个用来在代码段之间发送数据的消

息，那就考虑使用生产者 / 消费者队列。同样，不要让通用的信号只是用来协调对共享数据的访问。那种情况下，可使用锁。



AsyncManualResetEvent 类在 NuGet 包 Nito.AsyncEx 中。

参阅

8.8 节介绍了异步的生产者 / 消费者队列。

11.2 节介绍了异步锁。

11.3 节介绍了阻塞信号，用于在线程间发送通知。

11.5 限流

问题

有一段高度并发的代码，由于它的并发程度实在太高了，需要有方法对并发性进行限流。

代码并发程度太高，是指程序中的一部分无法跟上另一部分的速度，导致数据项累积并消耗内存。这种情况下对部分代码进行限流，可以避免占用太多内存。

解决方案

根据代码的并发类型，解决方法各有不同。这些解决方案都是把并发性限制在某个范围之内。响应式扩展有更多功能强大的方法可以选择，例如滑动时间窗口。5.4 节对 Rx 限流有更完整的介绍。

数据流和并行代码都自带了对并发性限流的方法：

```
IPropagatorBlock<int, int> DataflowMultiplyBy2()
{
    var options = new ExecutionDataflowBlockOptions
    {
        MaxDegreeOfParallelism = 10
    };

    return new TransformBlock<int, int>(data => data * 2, options);
}

// 使用 PLINQ
```

```
IEnumerable<int> ParallelMultiplyBy2(IEnumerable<int> values)
{
    return values.AsParallel()
        .WithDegreeOfParallelism(10)
        .Select(item => item * 2);
}

// 使用 Parallel 类
void ParallelRotateMatrices(IEnumerable<Matrix> matrices, float degrees)
{
    var options = new ParallelOptions
    {
        MaxDegreeOfParallelism = 10
    };
    Parallel.ForEach(matrices, options, matrix => matrix.Rotate(degrees));
}
```

并发性异步代码可以用 SemaphoreSlim 来限流：

```
async Task<string[]> DownloadUrlsAsync(IEnumerable<string> urls)
{
    var httpClient = new HttpClient();
    var semaphore = new SemaphoreSlim(10);
    var tasks = urls.Select(async url =>
    {
        await semaphore.WaitAsync();
        try
        {
            return await httpClient.GetStringAsync(url);
        }
        finally
        {
            semaphore.Release();
        }
    }).ToArray();
    return await Task.WhenAll(tasks);
}
```

讨论

如果发现程序使用的资源（例如 CPU 或网络连接）太多，说明可能需要使用限流了。需要牢记一点，最终用户的电脑性能可能不如开发者的电脑，因此限流得稍微严格一点，比限流不充分要好。

参阅

5.4 节介绍了在响应式代码中进行限流。

第12章
调度

代码必须在某个线程上运行。调度器（scheduler）是一个确定代码运行地点的对象。.NET 框架中有几种不同的调度器类型，并行和数据流代码也使用调度器，但方式有些细微的区别。

我建议大家尽量不要指定调度器，系统默认的调度器通常是最合适的。例如，异步代码中的 await 操作符会自动选择在当前上下文中恢复运行，除非用 2.7 节中描述的方法覆盖默认选项。类似地，响应式代码引发事件时用的默认上下文是很合理的，可以如 5.2 节描述的那样，用 ObserveOn 覆盖默认选项。

但是要让其他代码在指定的上下文（如 UI 线程上下文或 ASP.NET 请求上下文）中运行，就可以使用本章的调度技巧来调度代码。

12.1　调度到线程池

问题

指定一段代码在线程池线程中执行。

解决方案

绝大多数情况下可以使用 Task.Run，它用起来很简单。下面的代码阻塞一个线程池线程 2 秒钟：

```
Task task = Task.Run(() =>
{
```

```
        Thread.Sleep(TimeSpan.FromSeconds(2));
    });
```

Task.Run 也能正常地返回结果，能使用异步 Lambda 表达式。下面代码中 Task.Run 返回的 task 会在 2 秒后完成，并返回结果 13：

```
    Task<int> task = Task.Run(async () =>
    {
        await Task.Delay(TimeSpan.FromSeconds(2));
        return 13;
    });
```

Task.Run 返回一个 Task（或 Task<T>）对象，该对象可以被异步或响应式代码正常使用。

讨论

如果在 UI 程序中有很耗时的任务，但不能在 UI 线程中执行该任务，这时使用 Task.Run 就非常合适。例如，7.4 节中使用 Task.Run 把并行处理任务放到线程池线程。但不要在 ASP.NET 中使用 Task.Run，除非你有绝对的把握。在 ASP.NET 中，处理请求的代码本来就是在线程池线程中运行的，强行把它放到另一个线程池线程通常会适得其反。

Task.Run 完全可以替代 BackgroundWorker、Delegate.BeginInvoke 和 ThreadPool.QueueUserWorkItem。新写的代码都不要使用这些过时的技术，使用 Task.Run 的代码更利于正确编写和日后的维护。而且 Task.Run 能处理绝大多数使用 Thread 类的场景，大多数情况下可以用 Task.Run 来代替 Thread 类（有极少数例外情况，如单线程单元线程）。

并行和数据流代码默认在线程池中执行，因此 Parallel、Parallel LINQ 或 TPL 数据流库的代码通常不需要使用 Task.Run。

在进行动态并行开发时，一定要用 Task.Factory.StartNew 来代替 Task.Run。因为根据默认配置，Task.Run 返回的 Task 对象适合被异步调用（即被异步代码或响应式代码使用）。Task.Run 也不支持动态并行代码中普遍使用的高级概念，例如父 / 子任务。

参阅

7.6 节介绍如何用响应式代码调用异步代码（例如 Task.Run 返回的 Task 对象）。

7.4 节介绍如何异步地等待并行代码，最简单的办法是使用 Task.Run。

3.4 节介绍动态并行，即需要用 Task.Factory.StartNew 代替 Task.Run 的场景。

12.2 任务调度器

问题

需要让多个代码段按照指定的方式运行。例如让所有代码段在 UI 线程中运行，或者只允许特定数量的代码段同时运行。

本节介绍如何定义和构造这些代码段的调度器。后面两节介绍如何应用调度器。

解决方案

.NET 中有很多不同的类可以进行任务调度。本节重点讲 TaskScheduler，因为它便于移植，使用起来也相对容易。

最简单的 TaskScheduler 对象是 TaskScheduler.Default，它的作用是让任务在线程池中排队。在代码中很少会指定 TaskScheduler.Default，但是要特别注意这个问题，因为它是调度时最常用的默认值。Task.Run、并行、数据流的代码用的都是 TaskScheduler.Default。

可以捕获一个特定的上下文，然后用 TaskScheduler.FromCurrentSynchronizationContext 调度任务，让它回到该上下文：

```
TaskScheduler scheduler = TaskScheduler.FromCurrentSynchronizationContext();
```

这条语句创建了一个捕获当前 SynchronizationContext 的 TaskScheduler 对象，并将代码调度到这个上下文中。SynchronizationContext 类表示一个通用的调度上下文。.NET 中有几个不同的上下文，大多数 UI 框架有一个表示 UI 线程的 SynchronizationContext，ASP.NET 有一个表示 HTTP 请求上下文的 SynchronizationContext。

.NET 4.5 引入了另一个功能强大的类，即 ConcurrentExclusiveSchedulerPair，它实际上是两个互相关联的调度器。只要 ExclusiveScheduler 上没有运行任务，ConcurrentScheduler 就可以让多个任务同时执行。只有当 ConcurrentScheduler 没有执行任务时，ExclusiveScheduler 才可以执行任务，并且每次只允许运行一个任务：

```
var schedulerPair = new ConcurrentExclusiveSchedulerPair();
TaskScheduler concurrent = schedulerPair.ConcurrentScheduler;
TaskScheduler exclusive = schedulerPair.ExclusiveScheduler;
```

ConcurrentExclusiveSchedulerPair 的常见用法是用 ExclusiveScheduler 来确保每次只运行一个任务。ExclusiveScheduler 执行的代码会在线程池中运行，但是使用了同一个 ExclusiveScheduler 对象的其他代码不能同时运行。

ConcurrentExclusiveSchedulerPair 的另一个用法是作为限流调度器。创建的 Concurrent

ExclusiveSchedulerPair 对象可以限制自身的并发数量。这时通常不使用 Exclusive Scheduler：

```
var schedulerPair = new ConcurrentExclusiveSchedulerPair(TaskScheduler.Default,
    maxConcurrencyLevel: 8);
TaskScheduler scheduler = schedulerPair.ConcurrentScheduler;
```

注意，这种限流方式只是对运行中的代码限流，它与 11.5 节的逻辑上限流有很大区别。尤其要注意，正在等待一个操作完成的异步代码不属于运行中的代码。ConcurrentScheduler 对运行中的代码做限流。其他限流方法（如 SemaphoreSlim）在更高的层次做限流（即完整的异步方法）。

讨论

也许你已经注意到了，前面的示例代码中，在 ConcurrentExclusiveSchedulerPair 的构造函数中传入了 TaskScheduler.Default。这是因为 ConcurrentExclusiveSchedulerPair 其实是在一个已有的 TaskScheduler 对象基础上实现并发 / 独占逻辑的。

本节介绍了 TaskScheduler.FromCurrentSynchronizationContext，它可用于在捕获的上下文中执行代码。直接使用 SynchoronizationContext 在该上下文中执行代码也是可以的，但是我建议大家不要用这种做法。只要有可能，就要使用 await 操作符返回到隐式捕获的上下文，或者使用 TaskScheduler 的封装类。

在 UI 线程上执行代码时，永远不要使用针对特定平台的类型。WPF、Silverlight、iOS、Android 都有 Dispatcher 类，Windows 应用商店平台使用 CoreDispatcher，Windows Forms 有 ISynchronizeInvoke 接口（即 Control.Invoke）。不要在新写的代码中使用这些类型，就当它们不存在吧。使用这些类型会使代码无谓地绑定在某个特定平台上。SynchronizationContext 是通用的、基于上述类型的抽象类。

响应式扩展引入了一个更通用的调度器抽象类：IScheduler。Rx 调度器能够封装任何类型的其他调度器，TaskPoolScheduler 会封装任何 TaskFactory（TaskFactory 包含了 TaskScheduler）。Rx 还定义了 IScheduler 的一种可以手动控制的实现方式，用于测试。如果需要真正地使用调度器抽象类，建议大家使用 Rx 的 IScheduler。它具有良好的设计和定义，测试起来也很方便。不过大多数情况下并不需要调度器抽象类，也不需要使用早期的库（如任务并行库和 TPL 数据流），只要掌握 TaskScheduler 类就行了。

参阅

12.3 节介绍如何在并行代码中使用 TaskScheduler。

12.4 节介绍如何在数据流代码中使用 TaskScheduler。

11.5 节介绍高层次的逻辑限流。

5.2 节介绍响应式扩展的事件流调度器。

6.6 节介绍用于响应式扩展测试的调度器。

12.3 调度并行代码

问题

需要控制个别代码段在并行代码中的执行方式。

解决方案

创建了合适的 TaskScheduler 实例（见 12.2 节）后，可以把它放入 Parallel 类的方法参数中。下面的代码使用一系列矩阵的序列。启动一批并行循环，并且需要限制所有循环的总的并行数量，不管每个序列中有多少矩阵：

```
void RotateMatrices(IEnumerable<IEnumerable<Matrix>> collections, float degrees)
{
    var schedulerPair = new ConcurrentExclusiveSchedulerPair(
        TaskScheduler.Default, maxConcurrencyLevel: 8);
    TaskScheduler scheduler = schedulerPair.ConcurrentScheduler;
    ParallelOptions options = new ParallelOptions { TaskScheduler = scheduler };
    Parallel.ForEach(collections, options,
        matrices => Parallel.ForEach(matrices, options,
            matrix => matrix.Rotate(degrees)));
}
```

讨论

Parallel.Invoke 也能使用 ParallelOptions 实例，因此可以像 Parallel.ForEach 那样把 TaskScheduler 传入 Parallel.Invoke。在编写动态并行代码时，可以直接把 TaskScheduler 传入 TaskFactory.StartNew 或者 Task.ContinueWith。

没有什么办法能把 TaskScheduler 传入 PLINQ 代码。

参阅

12.2 节介绍常用的任务调度器以及如何选择调度器。

12.4 用调度器实现数据流的同步

问题

需要控制个别代码段在数据流代码中的执行方式。

解决方案

创建了合适的 TaskScheduler 实例（见 12.2 节）后，可以把它放入数据流块的参数中。下面的代码被 UI 线程调用时，会创建一个数据流网格。这个数据流网格将每个输入值乘以 2（用线程池），然后把结果添加到一个列表控件的项目中（在 UI 线程中）：

```
var options = new ExecutionDataflowBlockOptions
{
    TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext(),
};
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var displayBlock = new ActionBlock<int>(
    result => ListBox.Items.Add(result), options);
multiplyBlock.LinkTo(displayBlock);
```

讨论

如果要协调位于数据流网格中不同部位的块的行为，就非常需要指定一个 TaskScheduler。例如，可以用 ConcurrentExclusiveSchedulerPair.ExclusiveScheduler 来确保块 A 和块 C 永远不同时执行代码，而块 B 可以随时执行。

记住，TaskScheduler 的同步功能只有在代码运行时才起作用。例如对一个运行异步代码的执行块启用一个独占调度器，当它正在等待时，不被认为是在运行。

可以对任何类型的数据流块指定一个 TaskScheduler。即使一个块可能执行外来的代码（如 BufferBlock<T>），它仍需要做一些内部协调任务，并且会在内部任务中使用 TaskScheduler。

参阅

12.2 节介绍常用的任务调度器以及如何选择调度器。

第 13 章

实用技巧

本章我们来看几个编写并发程序时经常遇到的场景，以及处理此类场景需要用到的各种类和技术。这样的场景足够写满另外一本书，因此这里只选了几个最有用的来进行分析。

13.1 初始化共享资源

问题

程序的多个部分共享了一个资源，现在要在第一次访问该资源时对它初始化。

解决方案

.NET 框架中有一个专门用来解决这种问题的类：Lazy<T>。在构造这个类的实例时，用一个工厂委托（factory delegate）进行初始化。通过 Value 属性使这个实例变得可用。下面的代码演示了 Lazy<T> 类：

```
static int _simpleValue;
static readonly Lazy<int> MySharedInteger = new Lazy<int>(() => _simpleValue++);

void UseSharedInteger()
{
    int sharedValue = MySharedInteger.Value;
}
```

不管同时有多少线程调用 UseSharedInteger，这个工厂委托只会运行一次，并且所有线程都等待同一个实例。实例在创建后会被缓存起来，以后所有对 Value 属性的访问都返回同

一个实例（前面的例子中，MySharedInteger.Value 永远是 0）。

如果初始化过程需要执行异步任务，可以采用一个非常类似的方法。本例使用 Lazy
<Task<T>>：

```
static int _simpleValue;
static readonly Lazy<Task<int>> MySharedAsyncInteger =
    new Lazy<Task<int>>(async () =>
    {
        await Task.Delay(TimeSpan.FromSeconds(2)).ConfigureAwait(false);
        return _simpleValue++;
    });

async Task GetSharedIntegerAsync()
{
    int sharedValue = await MySharedAsyncInteger.Value;
}
```

本例中委托返回一个 Task<int> 对象，就是一个用异步方式得到的整数值。不管有多少代码段同时调用 Value，Task<int> 对象只会创建一次，并且每个调用都返回同一个对象。每个调用者可以用 await 调用这个 Task 对象，（异步地）等待它完成。

这种模式是可行的，但还有一点需要注意。这个异步的委托可能在任何调用 Value 的线程中运行，也就会在对应的上下文中运行。如果有几种不同类型的线程会调用 Value（例如一个 UI 线程和一个线程池线程，或者两个不同的 ASP.NET 请求线程），那最好让委托只在线程池线程中运行。这实现起来很简单，只要把工厂委托封装在 Task.Run 调用中：

```
static readonly Lazy<Task<int>> MySharedAsyncInteger = new Lazy<Task<int>>(
    () => Task.Run(
        async () =>
        {
            await Task.Delay(TimeSpan.FromSeconds(2));
            return _simpleValue++;
        }));
```

讨论

最后一个例子是 Lazy 对象异步初始化的通用模式，可惜有些繁琐。AsyncEx 库中有一个与 Lazy<Task<T>> 功能相似的类 AsyncLazy<T>，它会在线程池中执行工厂委托。它也可以直接进行 await，声明和使用的方法如下：

```
private static readonly AsyncLazy<int> MySharedAsyncInteger =
    new AsyncLazy<int>(async () =>
    {
        await Task.Delay(TimeSpan.FromSeconds(2));
        return _simpleValue++;
    });
```

```
public async Task UseSharedIntegerAsync()
{
    int sharedValue = await MySharedAsyncInteger;
}
```



AsyncLazy<T> 类在 NuGet 包 Nito.AsyncEx 中。

参阅

第 1 章介绍了 async/await 方式编程的基础知识。

12.1 节介绍了如何把任务调度到线程池。

13.2 Rx延迟求值

问题

想要在每次被订阅时就创建一个新的源 observable 对象。例如让每个订阅代表一个不同的 Web 服务请求。

解决方案

Rx 库有一个操作符 Observable.Defer，每次 observable 对象被订阅时，它就会执行一个委托。该委托相当于是一个创建 observable 对象的工厂。下面的代码中每次订阅 observable 对象，都会使用 Defer 调用一个异步方法：

```
static void Main(string[] args)
{
    var invokeServerObservable = Observable.Defer(
        () => GetValueAsync().ToObservable());
    invokeServerObservable.Subscribe(_ => { });
    invokeServerObservable.Subscribe(_ => { });

    Console.ReadKey();
}

static async Task<int> GetValueAsync()
{
    Console.WriteLine("Calling server...");
    await Task.Delay(TimeSpan.FromSeconds(2));
    Console.WriteLine("Returning result...");
    return 13;
}
```

代码的输出结果为：

```
Calling server...
Calling server...
Returning result...
Returning result...
```

讨论

应用程序的代码一般不会多次订阅一个 observable 对象，但有些 Rx 操作符会在内部多次订阅一个 observable 对象。例如一旦条件满足，`Observable.While` 操作符会重新订阅一个源序列。用 `Defer` 可以让 observable 对象在每次有新的订阅时就重新求值。如果需要刷新或更新 observable 对象的数据，就可以用这个方法。

参阅

7.6 节介绍如何把异步方法封装进 observable 对象。

13.3 异步数据绑定

问题

在异步地检索数据时，需要对结果进行数据绑定（例如绑定到 Model-View-ViewModel 设计模式中的 ViewModel）。

解决方案

如果使用属性进行数据绑定，这个属性必须立即同步地返回某种结果。如果需要异步地确定实际值，那可以先返回一个默认值，以后用正确值来更新这个属性。

需要注意的是，异步操作的结果可能是成功，也可能是失败。因为编写的是 ViewModel，在出错的情况下也可以用数据绑定的方式来更新 UI。

可以使用 `AsyncEx` 库中的 `NotifyTaskCompletion` 类：

```
class MyViewModel
{
    public MyViewModel()
    {
        MyValue = NotifyTaskCompletion.Create(CalculateMyValueAsync());
    }

    public INotifyTaskCompletion<int> MyValue { get; private set; }
```

```
    private async Task<int> CalculateMyValueAsync()
    {
        await Task.Delay(TimeSpan.FromSeconds(10));
        return 13;
    }
}
```

可以绑定到 INotifyTaskCompletion<T> 属性中的各种属性，如下所示：

```
<Grid>
    <Label Content="Loading..."
        Visibility="{Binding MyValue.IsNotCompleted,
            Converter={StaticResource BooleanToVisibilityConverter}}"/>
    <Label Content="{Binding MyValue.Result}"
        Visibility="{Binding MyValue.IsSuccessfullyCompleted,
            Converter={StaticResource BooleanToVisibilityConverter}}"/>
    <Label Content="An error occurred" Foreground="Red"
        Visibility="{Binding MyValue.IsFaulted,
            Converter={StaticResource BooleanToVisibilityConverter}}"/>
</Grid>
```

讨论

也可以自己编写数据绑定的封装类代替 AsyncEx 库中的类。下面的代码介绍了基本思路：

```
class BindableTask<T> : INotifyPropertyChanged
{
    private readonly Task<T> _task;

    public BindableTask(Task<T> task)
    {
        _task = task;
        var _ = WatchTaskAsync();
    }

    private async Task WatchTaskAsync()
    {
        try
        {
            await _task;
        }
        catch
        {
        }

        OnPropertyChanged("IsNotCompleted");
        OnPropertyChanged("IsSuccessfullyCompleted");
        OnPropertyChanged("IsFaulted");
        OnPropertyChanged("Result");
    }

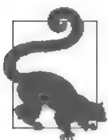
    public bool IsNotCompleted { get { return !_task.IsCompleted; } }
    public bool IsSuccessfullyCompleted
```

```
    { get { return _task.Status == TaskStatus.RanToCompletion; } }
    public bool IsFaulted { get { return _task.IsFaulted; } }
    public T Result
    { get { return IsSuccessfullyCompleted ? _task.Result : default(T); } }

    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
            handler(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

注意，这里有一个空的 catch 语句，其目的是明确地捕获所有的异常，并且通过数据绑定来处理那些情况。另外，这里显然不能使用 ConfigureAwait(false)，因为 PropertyChanged 事件会在 UI 线程中引发。



NotifyTaskCompletion 类在 NuGet 包 Nito.AsyncEx 中。

参阅

第 1 章介绍了 async/await 方式编程的基础知识。

2.7 节介绍了如何使用 ConfigureAwait。

13.4 隐式状态

问题

程序中有一些状态变量，要求在调用栈的不同位置都可以访问。例如，在记录日志时要使用一个当前操作的标识符，但是又不希望把它作为参数添加到每一个方法中。

解决方案

最好的做法是在方法中增加参数，存储在类的成员变量中，或者使用依赖注入来为代码的各个部分提供数据。但是在有些情况下，这么做会使代码变得过于复杂。

使用 .NET 中 CallContext 类的 LogicalSetData 和 LogicalGetData 方法，可以为一个状态命名，并把它放在一个逻辑“上下文”中。用完这个状态后，可以调用 FreeNamedDataSlot

把它从上下文中移除。以下代码演示了如何用这些方法来设置操作标识符，然后在日志方法中使用：

```
void DoLongOperation()
{
    var operationId = Guid.NewGuid();
    CallContext.LogicalSetData("OperationId", operationId);

    DoSomeStepOfOperation();

    CallContext.FreeNamedDataSlot("OperationId");
}
void DoSomeStepOfOperation()
{
    // 在这里记录日志。
    Trace.WriteLine("In operation: " +
        CallContext.LogicalGetData("OperationId"));
}
```

讨论

可以在 async 方法中使用逻辑调用上下文（logical call context），但仅限于 .NET 4.5 和更高版本。如果在 .NET 4.0 和 NuGet 包 Microsoft.Bcl.Async 上使用，代码能够编译通过但不会正确地运行。

在逻辑调用上下文中，应该只存储不可变数据。如果要修改逻辑调用上下文中的数据，应该重新调用 LogicalSetData 来覆盖已有的数据。

逻辑调用上下文的效率不是特别高。我建议大家只要有可能，就在方法中添加参数或者把数据存储在类的成员中，而不是用隐式的逻辑调用上下文。

在编写 ASP.NET 程序时可考虑使用 HttpContext.Current.Items，它的功能和 CallContext 一样，但效率更高。

参阅

第 1 章介绍了 async/await 方式编程的基础知识。

第 8 章介绍了几种不可变集合，可以用它们来存储复杂的数据，并将其作为隐式状态。

封面介绍

本书封面的动物是一只麝香猫，也称亚洲麝香猫。抛开它们的拉丁文名字，同其他哺乳动物一样，它们也有两种不同的性别，并不是雌雄同体。麝香猫大部分时间都是独居，只有在繁殖季节，雄性和雌性才会聚在一起。它们的原生地是东南亚和印度尼西亚群岛，近年也被引进到了日本和小巽他群岛。

麝香猫是一种小型的毛皮动物，身长最长可达 53 厘米，体重最多可重 5 千克。它们通常有白色或灰色的斑纹，头部有点像浣熊。与其他灵猫类动物不同，麝香猫的尾巴是没有圆环的。麝香猫抵御捕食者的最佳武器就是，受到威胁时肛门处的气味腺释放出的难闻分泌物。发达的嗅觉也有助于它们交配，雄性和雌性会利用气味在森林里找到对方。

麝香猫是夜间活动的杂食动物，它们会到处散播水果种子，在维护森林的生物多样性上起了重要作用。它们特别喜欢喝棕榈花的汁液，这种汁液在发酵后就成为甜美的棕榈酒，它们因此赢得了“棕榈酒猫”的绰号。在一些地方，特别是中国南部，人们为了吃肉而捕杀麝香猫。但其实对麝香猫种群最大的威胁，是人们为了生产猫屎咖啡而捕捉野生麝香猫。猫屎咖啡是用经过麝香猫消化并排泄的咖啡豆制作的，传统制作过程用到了野生动物的粪便。喝这种咖啡的人越来越多，导致麝香猫被捕捉并关在大农场的小笼子里，只能吃咖啡豆，不能运动，也不能到野外活动。Tony Wild 是负责把猫屎咖啡传播到西方的营销经理，出于动物保护的原因，现在他反对这种做法，并发起了一场名为“停用粪便”（Cut the Crap）的运动，以停止它的使用。

封面图片取自 Lydekker 的《皇家自然史》。

关注图灵教育 关注图灵社区

iTuring.cn

在线出版 电子书《码农》杂志 图灵访谈……



QQ联系我们

读者QQ群：218139230



微博联系我们

官方账号：@图灵教育 @图灵社区 @图灵新知
市场合作：@图灵袁野
写作本版书：@图灵小花
翻译英文书：@李松峰 @朱巍ituring @楼伟珊
翻译日文书或文章：@图灵乐馨
翻译韩文书：@图灵陈曦
电子书合作：@hi_jeanne
图灵访谈/《码农》杂志：@李盼ituring
加入我们：@王子是好人



微信联系我们



图灵教育
turingbooks



图灵访谈
ituring_interview

O'REILLY®

C#并发编程经典实例

并发编程在响应式和可扩展的应用开发中得到了日益广泛的应用。但并发编程的难度曾经非常大，令众多开发人员望而却步。今天，很多更高层抽象的现代程序库的出现，大大降低了并发编程的难度。本书使用.NET 4.5和C# 5.0中的语言特性，展示并行处理和异步编程技术。

本书既是一本入门指导书，也是一本快捷参考书，它示例丰富、结构独特，70多个源代码示例，完整的“问题－解决方案－讨论”模式，逐渐深入又自成一体。你可以循序渐进地学习本书内容，也可以直接查阅对应的示例，迅速解决手头的问题。

本书主要内容：

- 面向异步编程的async和await
- 使用TPL（任务并行库）
- 创建数据流管道的TPL Dataflow库
- 基于LINQ的Reactive Extensions
- 为并发代码编写单元测试
- 并发方法之间的互操作
- 不可变、线程安全和生产者/消费者集合
- 并发代码中的取消功能支持
- 支持异步的面向对象编程
- 线程同步访问数据

“涵盖各种并发编程技术，本书体例必然成就其为现代.NET并发技术的理想参考书。”

——Jon Skeet
谷歌高级软件开发工程师，
StackOverflow排名第一
的杰出程序员，
著有《深入理解C#》

“让普通人利用大规模并行能力是计算领域的一大趋势。与以前相比，开发人员已经能更好地掌握并发技术，但要把并发讲清楚对很多人仍然是一项巨大的挑战。Stephen专注于这个领域，通过这本易读、完整的参考手册，帮助我们更好地理解并发、线程、反应式编程模型、并行等主题。”

——Scott Hanselman
微软ASP.NET及
Azure Web Tools首席项目经理

Stephen Cleary C# MVP，知名软件开发人员，在C#、C++、JavaScript等方面均有丰富的经验。1998年起成为专业软件开发人员，涉猎广泛，从ARM固件到Azure样样精通。他从最初的Boost C++库开始就在为开源软件做贡献，并且发布了几个他自己的库和工具。Stephen喜欢演讲和写作，在其个人网站http://stephencleary.com/上，有大量受欢迎的博客文章以及开源库和应用。

封面设计：Randy Comer　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机/程序设计/C#

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



ISBN 978-7-115-37427-1
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks